Final Report on
Collaborative Software Engineering Tools Workshop and Follow-Up

Project NAG2-1555

Edited by

David F. Redmiles, PhD

Institute for Software Research and
Department of Informatics
University of California, Irvine
Irvine, CA 92697-3425 USA
redmiles@ics.uci.edu

ISR Technical Report # UCI-ISR-03-14

December 2003

Abstract: This report documents the project "Collaborative Software Engineering Tools Workshop and Follow-Up." Under this project, a workshop was held at NASA/Ames on August 5 and 6, 2002. Additional research followed up on the workshop. Hence, the report contains these two components: materials from the workshop and a series of research papers that document our follow-up activities.

The workshop brought together technical staff of NASA/Ames and faculty and staff researchers from University of California's Institute for Software Research (ISR). The goal of the workshop was to generate a joint understanding of collaborative software engineering tools informed from four perspectives: 1) technology, 2) theory, 3) field studies, and 4) specific NASA problems.

The follow-up work included an intern working at NASA and providing some analysis of observations during the course of that work. The analysis was carried out collaboratively between personnel at NASA/Ames and UCI/ISR. Additional experimental software development was performed at UCI examining the role and architecture of event notification servers and awareness. Also, some initial explorations about extending field study methods were done.

Since the workshop, a web site has been maintained at
http://www.isr.uci.edu/events/NASA-Workshop/

Table of Contents

Part 1: Workshop Materials

Agenda

Monday, August 5

10:30 - 12:00 **Scope of Collaborative Software Engineering**

Introductions all around (15 mins)

Introductory Remarks
David Redmiles, UCI/ISR Faculty and
John Penix, Computer Scientist, NASA Ames

Challenges in Distributed Collaborative Space Mission Design
Gloria Mark, ISR Faculty

ScienceOrganizer: A Collaborative Information Management Tool for Scientific Teams
Richard M. Keller, Senior Computer Scientist, NASA Ames

Discussion (15 mins)

12:00 - 1:00 **Lunch**

1:00 - 2:15 **Quantification and Visualization**

Palantír: Increasing Awareness among Distributed Workspaces
André van der Hoek, ISR Faculty

Visualizing Software Instability
Jennifer Bevan, UC Santa Cruz Graduate Student

Source Code Instrumentation and Quantification of Events
Robert Filman, Computer Scientist, NASA Ames

Visualization of Software and Development
Paul Dourish, ISR Faculty

Discussion (15 mins)

2:15 - 2:30 **Break**

2:30 - 3:45 **Collaboration Studies and Tools**

Exploring the Relationship between Project Selection and Requirements Analysis
Mark Bergman, ISR Graduate Student

Past and Future of Postdoc
Chris Knight, NASA Ames

A Field Study of Collaborative Software Development Teams
Cleidson de Souza, ISR Graduate Student, NASA Ames Summer Intern

How do we go where no one has gone before?
Issues in the development of Autonomous Operations for Space
Kanna Rajan, NASA Ames

Discussion (15 mins)

3:45 - 4:00 **Break**

4:00 - 5:00 **Architecture and Synthesis**

US/France Coalition Warfare as a Model of Dynamic Architectures for Cross-Organizational Software Engineering
Richard N. Taylor, ISR Director and Faculty
(no slides available)

Formal Peer Inspection Information Architecture
Gilda Pour, Faculty, San Jose State University

Software Design Modelling and Code Generation Tools
Jon Whittle, Computer Scientist, NASA Ames
(no slides available)

Discussion (15 mins)

Tuesday, August 6

9:30 - 11:30 Event Infrastructure and Wrap up

From Simulation to Implementation - An overview of the Brahms Research and its application to Work Practice Analysis and Software Agents
Maarten Sierhuis, Senior Scientist, NASA Ames

Event-notification and Messaging Architectures for Real-time Science Coordination
Elias Sinderson, UC Santa Cruz Graduate Student, NASA Ames Summer Intern

Using Event Notification Servers to Support Awareness
David Redmiles, ISR Faculty

Discussion on Common Themes (1 hr 15 mins)
Moderated by David Redmiles and John Penix

11:30 - 12:30 Lunch

After lunch, a subset of people will meet to make future plans. Subset includes David Redmiles, John Penix, Michael Kantor, Susan Knight, Debra Brodbeck, at least 1 more UCI faculty, and 1 or more NASA or JPL people.

Participants

NASA Ames

- Martha DelAlto, md@ptolemy.arc.nasa.gov
- David Bell, dbell@arc.nasa.gov
- Robert Filman, Computer Scientist, rfilman@arc.nasa.gov
- Rich Keller, Senior Computer Scientist, rkeller@arc.nasa.gov
- Chris Knight, Computer Scientist, cknight@mail.arc.nasa.gov
- Jeff Lee, jmlee@arc.nasa.gov
- Kenneth I. Laws, klaws@email.arc.nasa.gov
- Masoud Mansouri-Samani, masoud@email.arc.nasa.gov
- Larry Markosian, zaven@email.arc.nasa.gov
- Peter Mehlitz, pcmehlitz@email.arc.nasa.gov
- Owen O'Malley, Computer Scientist, owen@email.arc.nasa.gov
- Joan Pallix, jpallix@mail.arc.nasa.gov
- John Penix, Computer Scientist and Workshop Organizer, jpenix@ptolemy.arc.nasa.gov
- Tom Pressburger, ttp@email.arc.nasa.gov
- Kanna Rajan, Senior Researcher, kanna@ptolemy.arc.nasa.gov
- David Roland, droland@mail.arc.nasa.gov
- John Shupe, jshupe@mail.arc.nasa.gov
- Maarten Sierhuis, Senior Scientist, msierhuis@mail.arc.nasa.gov
- Jon Whittle, Computer Scientist, jonathw@ptolemy.arc.nasa.gov

ISR/UCI

- Mark Bergman, Graduate Student, mbergman@ics.uci.edu
- Debra Brodbeck, ISR Technical Relations Director, brodbeck@uci.edu
- Eric Dashofy, Graduate Student, edashofy@ics.uci.edu
- Cleidson R. B. de Souza, Graduate Student, cdesouza@ics.uci.edu (currently on summer Internship at NASA Ames)
- Paul Dourish, Faculty, jpd@ics.uci.edu
- Roberto S.S. Filho, Graduate Student, rsilvafi@ics.uci.edu
- Michael Kantor, Post-Doctoral Researcher, mkantor@ics.uci.edu
- Susan Knight, ISR Corporate Relations Officer, sknight@uci.edu
- Gloria Mark, Faculty, gmark@ics.uci.edu

- David Redmiles, Faculty and Workshop Organizer, redmiles@ics.uci.edu

- Richard Taylor, ISR Director and Faculty, taylor@uci.edu
- André van der Hoek, Faculty, andre@ics.uci.edu

ISR/UC Santa Cruz

- Jennifer Bevan, Graduate Student, jbevan@cse.ucsc.edu
- Elias Sinderson, Graduate Student and NASA Ames Summer Intern (2002), elias@cse.ucsc.edu

Other

- Dale Martin, Clifton Labs, Inc., dmartin@cliftonlabs.com
- Lantz Moore, Senior Software Engineer, Clifton Labs, Inc., lmoore@cliftonlabs.com
- Gilda Pour, Faculty, San Jose State University and NRC Research Associate, NASA Ames, gpour@email.sjsu.edu
- Jason Robbins, Collab.Net, Inc., jrobbins@collab.net

Logistics

- Sessions
  - Usually 1 hour 15 minutes
  - Talks are 15 minutes including clarifying questions
  - 15 minutes of discussion
  - Break for more discussion

ISR
Institute for Software Research
University of California, Irvine

Workshop on Collaborative Software Engineering Tools

John Penix
Computer Scientist
NASA Ames

David Redmiles
Associate Professor
UC Irvine / ISR

Debra Brodbeck
Technical Relations Director
UC Irvine / ISR

ISR University of California Institute for Software Research

To advance software and information technology through research partnerships.

ISR is the only organization focused on Software Research within the University of California system.

http://www.isr.uci.edu/

Goals and Scope of the Workshop

- Understanding Collaboration and Software Tools
  - Real world setting informed by research
  - Mutual Education
- Aspects of the Scope
  - Difficulties in studying and supporting collaboration
  - NASA Problems and Opportunities
  - UCI Research Technology, Studies, and Opportunities

Research Emphases

http://www.isr.uci.edu/research.html

- **Analysis and Testing**
  Debra J. Richardson David S. Rosenblum
- **Computer Supported Cooperative Work**
  Mark Ackerman Paul Dourish
  Alfred Kobsa Gloria Mark
  David Redmiles
- **Configuration Management**
  André van der Hoek Jim Whitehead
- **Development Environments**
  David F. Redmiles David S. Rosenblum
  Debra J. Richardson Richard N. Taylor
  André van der Hoek
- **Human-Computer Interaction**
  Paul Dourish Alfred Kobsa
  Simon Penny David F. Redmiles
- **Hypermedia**
  Alfred Kobsa David F. Redmiles
  Richard N. Taylor Jim Whitehead
- **Internet-scale Event Notification**
  Alfonso Fuggetta David S. Rosenblum
- **Open Source**
  Mark Ackerman Alfonso Fuggetta
  Les Gasser Walt Scacchi
  Jim Whitehead
- **Software Acquisition and Electronic Commerce**
  Walt Scacchi
- **Software Architecture**
  David F. Redmiles David S. Rosenblum
  Richard N. Taylor Scott Samuelson
  André van der Hoek
- **Software Engineering Education**
  André van der Hoek
- **Software Understanding**
  Paul Dourish David F. Redmiles

Within the University Context…

- 15 Faculty Associates
- 40 Ph.D. students
- 2 Visiting Scholars
- ~9 Staff members (technical and administrative)

How can companies get involved?

- Research Partnerships enabling Collaborative Research Teams
  - External funding from Government or Industry sources (NSF, DoD/DARPA, NASA, etc.)
  - Internal funding from Corporate Partner
- Research Affiliates
  - Corporate sponsored research project
  - Graduate student research fellowship
  - ISR research presentations on-site
- Research Sponsors
  - Corporate affiliation on ISR Web Site and Events

Faculty

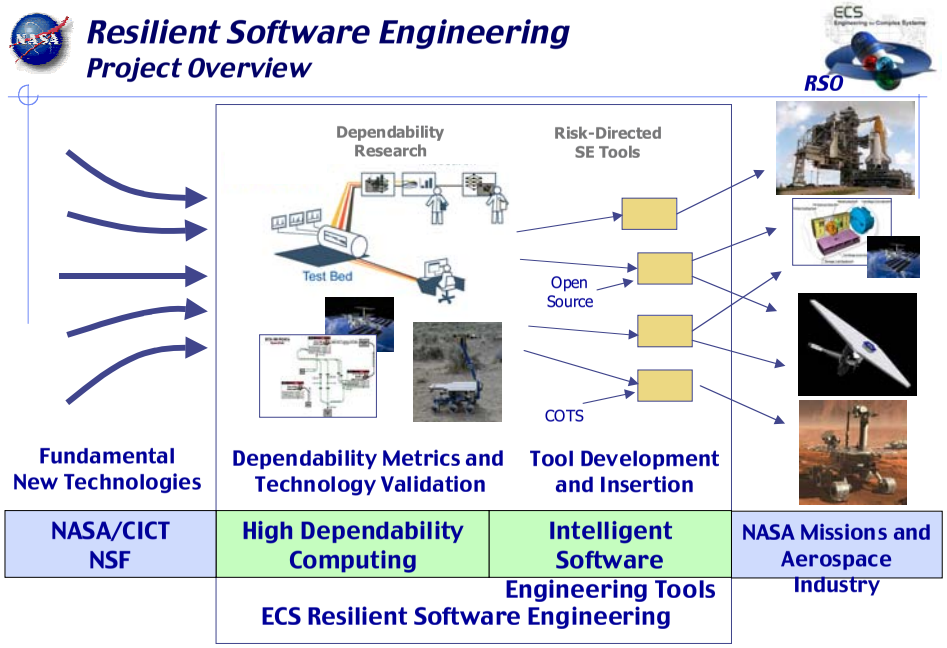
- **Richard N. Taylor**
  UCI/ICS and Director of ISR
- **Mark Ackerman**
  University of Michigan
- **Paul Dourish**
  UCI/ICS
- **Alfonso Fuggetta**
  Politecnico di Milano
- **Les Gasser**
  University of Illinois at Urbana-Champaign
- **Alfred Kobsa**
  UCI/ICS
- **Gloria Mark**
  UCI/ICS
- **Simon Penny**
  UCI/School of the Arts and the School of Engineering.
- **David F. Redmiles**
  UCI/ICS
- **Debra J. Richardson**
  UCI/ICS and ICS Dept. Chair
- **David S. Rosenblum**
  UCI/ICS
- **Scott Samuelsen**
  UCI/School of Engineering and Director, Advanced Power and Energy Program (APEP)
- **Walt Scacchi**
  UCI/ICS
- **André van der Hoek**
  UCI/ICS
- **E. James Whitehead**
  UC Santa Cruz

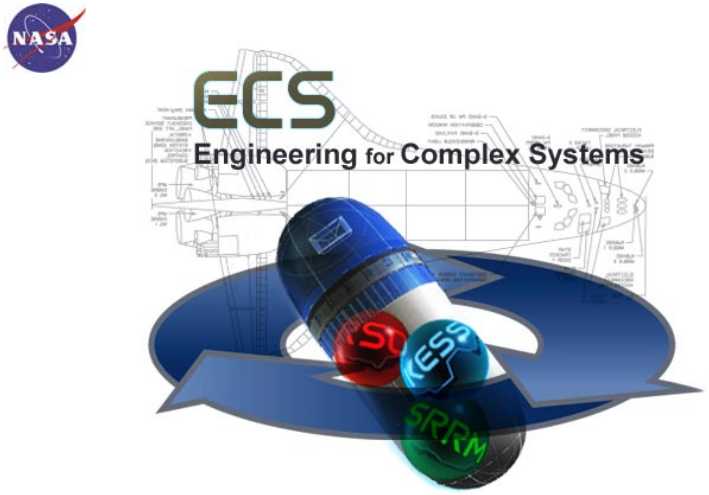
For More Information

Debra A. Brodbeck
Technical Relations Director
brodbeck@uci.edu
(949) 824-2260

Dr. Susan J. Knight
Corporate Relations Officer
sknight@uci.edu
(949) 824-5927

http://www.isr.uci.edu/





Collaborative Software Engineering Tools Workshop

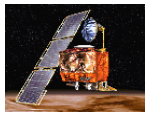
Dr. John Penix

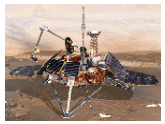
Intelligent Software Engineering Tools Goal

- **Reduce mission critical risks by developing tools and methods to identify and eliminate software errors**

Mishap Cause Classification: 1/3 aerospace mishaps are software related

Mars Climate Orbiter





Mars Polar Lander

Ariane 5

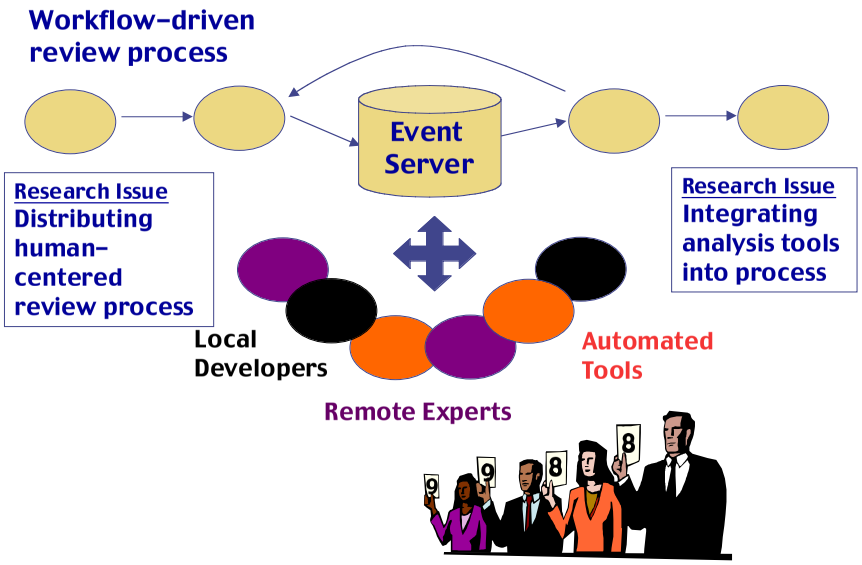
- **Sources of critical software risk:**
  - **Misunderstanding requirements and hardware software interface**
  - **Poor communication between teams**
  - **Insufficient design and testing**
  - **Inadequate or inappropriate methods and processes**

*Motivation**

- **Over 10% of NASA's civil servant and contractor workforce spend the majority of their time managing, developing, assuring, verifying, and/or maintaining software**
- **NASA has in operation use (and maintains) at least 200 million lines of source code**
- **Over $1 billion dollars of NASA's annual $15 billion budget is software cost**

* Based on estimates extrapolated from a 1993 study – source NASA Chief Engineer's Office

Distributed Collaborative Software Reviews



Intelligent Software Engineering Tools Approach

- Mature advanced modeling and analysis tools
  - Advanced Software Verification and Testing
  - Integrated Formal/Informal Requirements Engineering
- Integrate and leverage state of art tool technology
  - Commercial and open source tools
  - Distributed collaboration frameworks
- Work with missions to infuse tools into specific processes:
  - Add early lifecycle requirements analysis capabilities
  - Improve testing effectiveness
  - Enable tool-supported, distributed code reviews

Workshop Goal

- Improve this presentation!
  - What are NASA's problems?
  - What are some potential solutions?
  - What technology do we have that can play a role?
  - What research needs to be done?
  - How do we do that research?

Collaborative Software Engineering Tools

- Problem: Miscommunication between teams is a common source of critical errors
  - NASA software is often developed by distributed multidisciplinary teams, compounding this problem
  - Existing software engineering tools do not provide strong support for collaboration
- Solution: Insert advanced tools into NASA mission processes by integrating with collaborative frameworks:
  - Integration of the Verification and Testing Tools into a collaborative environment to support collaborative software design and code reviews (ARC)

Two trends: distributed and collocated design

- Collaborative design across distance
  - *Examples*: CAD shared app: Telefly system, "round-the-clock" software development
  - *Empirical studies*: coordination problems: transitioning, integration, communication (Herbsleb et al., 1999; Grinter, 1998)
- Collaborative designing in same physical environment
  - *Examples*: new generation of electronic meeting rooms (e.g. i-land: Streitz, 1999; interactive workspaces: Stanford; Discovery Collaboratory: Arias et al.)
  - *Empirical studies*: radical collocation (Teasley et al., 2000; 2002; Mark, 2002)

Challenges in Collaborative Space Mission Design

Gloria Mark
University of California, Irvine

Dynamic Group Work Structures

- Framework for considering group work often assumes "stable" team membership/structures
- Distributed and collocated group work is far more complex:
  - team boundaries are fuzzy
  - people belong to multiple teams, working spheres
  - multiple roles
  - social relationships are dynamic
- In "warrooms", even locale cannot define a team structure

My Research Interests

- Distributed and collocated group design work
- How can the affordances of a "warroom" influence collaborative design?
- What technologies can benefit collaborative design for collocated and distributed work: e.g. life-size, wall-size HDTV
- What effects on design does group-to-group distributed collaboration have?

Example of collocated design work with dynamic networks

- In 1995, Team X formed at the JPL to serve as internal consultants to NASA in designing new space mission proposals, e.g. Mars Probe
- Team X designs a complex space mission in about nine hours
- How can physical collocation *and* technology together enable a team to produce a space mission proposal in such a remarkably short time?

Some Background Theory

- Social worlds (Strauss, Shibutani, Giddens)
- Co-presence (Giddens)
- Interaction when collocated: monitoring, adjusting behaviors, common use of artifacts, etc.
  - (e.g. Heath and Luff, Suchman, Harper et al., Robertson, Rouncefield et al., Schmidt and Wagner)
- Social networks

Methodology

First study:

- Fieldwork observing warroom for three months
  - Sidebar conversations coded
  - Seventeen in-depth semi-structured interviews
  - Artifacts collected
  - HDTV experiment: videotaping, questionnaires, group interview

Current study:

- Fieldwork observing remote sites
  - video & audio tapes of each remote site, wave files of remote conversations

Dynamic networks within a group

- Some types of social networks
  - intensional networks (Nardi et al., in press)
  - "knots" (Engestrom et al, 1999)
  - actor-network theory (Latour, 1996)
  - coalitions (Zager, 2000)
  - virtual community networks (Wellman, 1998)
- Also:
  - networks to exchange and process information even when people are collocated, also distributed
- Analysis: who interacts with who, in main and sidebar channels of communication

Example: Initiating Networking for Spontaneous Sidebar

- Power to Config. Graphics: Can we get any power during the flight? Will the cells be pointed out? Otherwise we'll need big monster batteries.
- Mission Design, Team Leader, Instruments are speaking across room to each other
- Structures overhears Power and Config. Graphics and joins them
- ACS overhears and joins conversation from across the room.
- .....

External Representations Used in the Warroom

Representation	Creator/Driver	H: Function
Individual workstations	Team member	Monitor others' work
Publish-subscribe	Entire team	Info flow
Spreadsheet	Entire team	Focusing agent
Orbit visualization program	Team leader	Visualizing information
Public display	Team leader	Shared view
Paper whiteboards	Team member	Visualizing information

An Exploratory Study Using Life-size HDTV with Team X

- Large 128" x 72" screen showing HDTV as a "window" to show activity between rooms + audio
- Team X split into two rooms
- Real space shuttle mission proposal
- Telephones, with phone numbers, to support sidebars
- Day 1: audio directly sent in, video sent through Gigabit Ethernet (.8 second lag)
- Day 2: both audio and video sent through Gigabit Ethernet (degraded audio)

Social networking: sidebars

- Avg. number coded during three-hour session: 98 (large variability)
- Have lasted from few seconds to 53 minutes
- Avg. engineer speaks 20 minutes in sidebar, range is 7-110 minutes in a three-hour session.
- Sidebars used to process information from spreadsheet: question assumptions, negotiate, find other options, etc.

Current Work: Group-to-Group Distributed Collaborative Design

- Whereas distributed teams might be considered a "sphere of work", what happens when different "spheres of work" collaborate?
- Studying JPL, Marshall, Glenn, Sandia, began April 2002
- Four focii:
  - Requirements: conception as well as function
  - Information flow
  - Social networking
  - Technology use

The Potential of High Telepresence

- Video used as means for observing activity in remote room: not as good for supporting "networking"
  - < 20% of the time, video used for sidebars
  - *"Difficult to hold a local sidebar without disturbing people in other room"*
  - A learning curve may exist
  - Requirements for sidebars across distance:
    - Need to understand who to speak with
    - Need seamless way to connect

Opportunities for New Technologies to Support Distributed Work

- How can we leverage people's ability to monitor others' work?
- Seamless support for sidebar conversations (intentional and spontaneous) without overload of information
- External representations that capture the *design rationale*, not just the result
- .....still working...

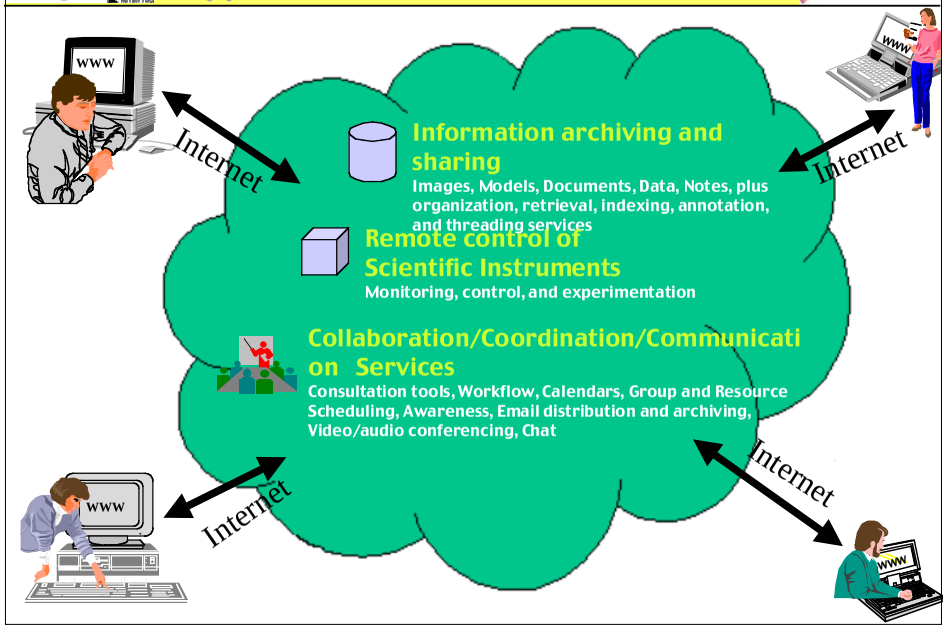
Networks at work in collocated environments

- Networks in collocated work easily break down over distance with the wrong technology support
- Delicate balance of automation and human processing
- Too much automation eases load, but may remove opportunity for creativity in design
- Flexibility is key: to move back and forth between electronic and social network

Thanks to....

- Paul DeFlorio, Bob Oberto, Rebecca Wheeler, JPL
- Team X
- Current study: Steve Abrams, Doug Grimes, Nayla Nassif

ScienceDesk Project: Infrastructure support for distributed scientific teams



ScienceDesk
Information Sharing for Scientific Project Teams

Collaborative Software Engineering Workshop

ScienceOrganizer:
A Collaborative Information Management Tool for Scientific Teams

Richard M. Keller, Ph.D.

Information Sharing and Integration Group
Collaborative and Assistant Systems Tech Area
Computational Sciences Division
NASA Ames Research Center

Research Areas

- **Scientific Knowledge Management**: capture, preservation, traceability of scientific knowledge
- **Intelligent Information Access**: intelligent indexing, visualization, and navigation
- **Collaboratories**: asynchronous and synchronous collaborative scientific teamwork
- **Agent-assisted Remote Experimentation**: intelligent monitoring and control

http://sciencedesk.arc.nasa.gov

ScienceDesk Project Staff

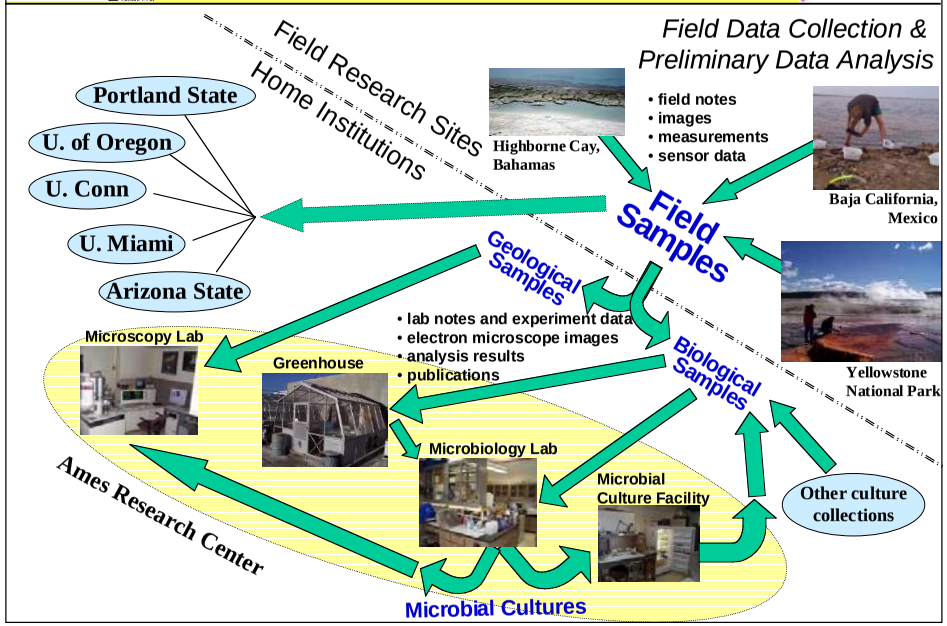
Rich Keller
Shawn Wolfe
David Hall, QSS
Robert Carvalho
Steve Rich, SAIC
Deepak Kulkarni
Dan Berrios, RIACS
Sergey Yentus, QSS
Keith Swanson
Ian Sturken, QSS
Ling-Jen Chiang, QSS
David Nishikawa
Linda Andrews, RIACS

Computational Sciences Division
NASA Ames Research Center

What is ScienceOrganizer?

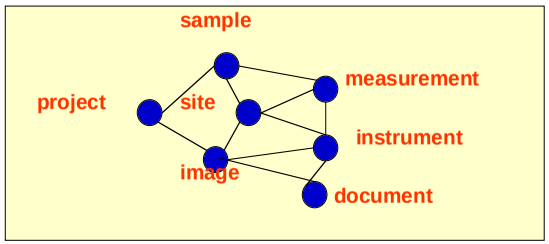
- **An information repository / digital library for distributed scientific project teams:** stores heterogeneous project information products -- *images, datasets, documents, and various types of scientific records* *(describing samples, field sites, measurements, instruments, microbial cultures, etc.)*
- **A hybrid tool combining the functionality of:**
  - a database
  - a document-sharing system
  - a hypermedia information space
  - a semantic network
- **Features cross-linkage:** enables rapid access to interrelated information
- **A "project memory" system:** tracks history of project team's fieldwork, labwork, and associated data collection activities

Motivation: Distributed Scientific Field and Lab Work



The ScienceOrganizer "Project Information Web"

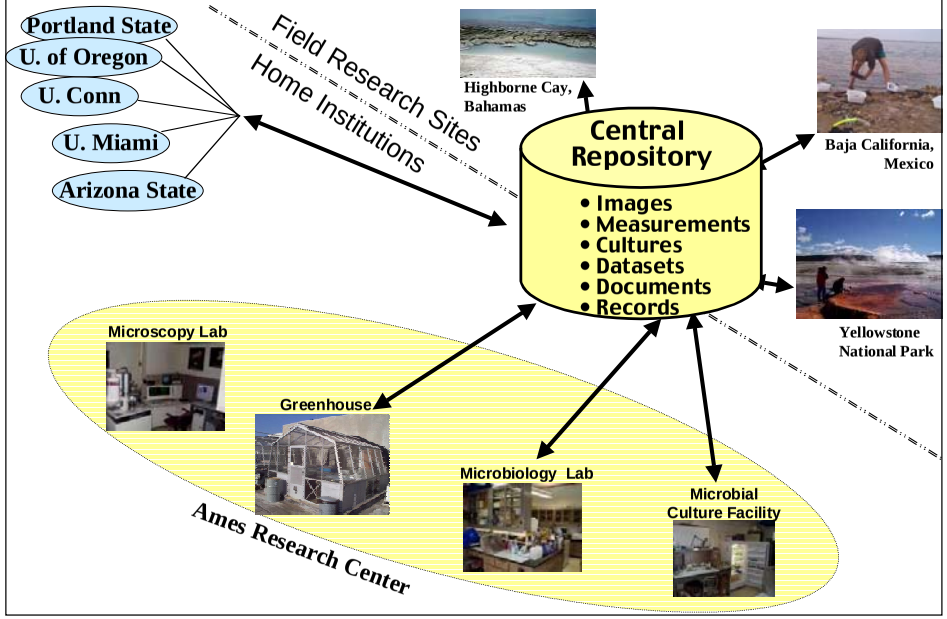
ScienceOrganizer maintains project information in an interconnected network or "information web"



- Nodes: *information resources*
- Links: *relationships among resources*

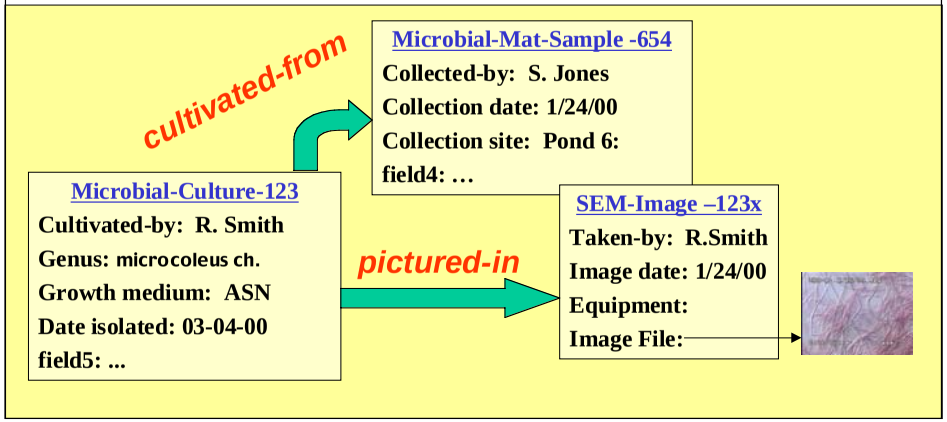
Semantic hypermedia system

ScienceOrganizer Information Repository



Relationships among Resources ("links")

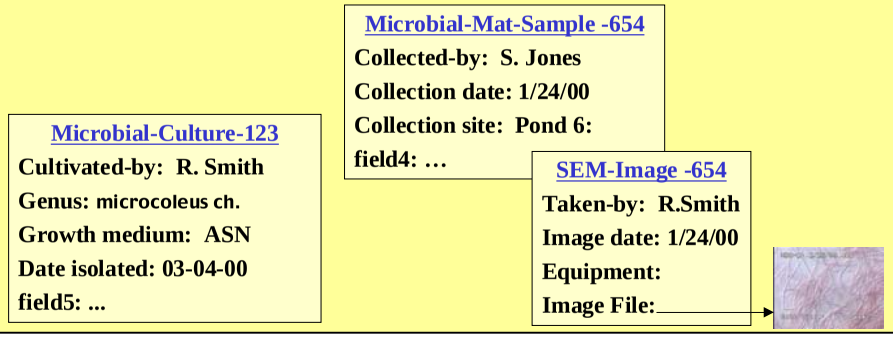
- Information Resources are interrelated by means of named links that characterize the nature of the relationship
- Relationships are customized to a project team based on an analysis of the information resources in the scientific domain



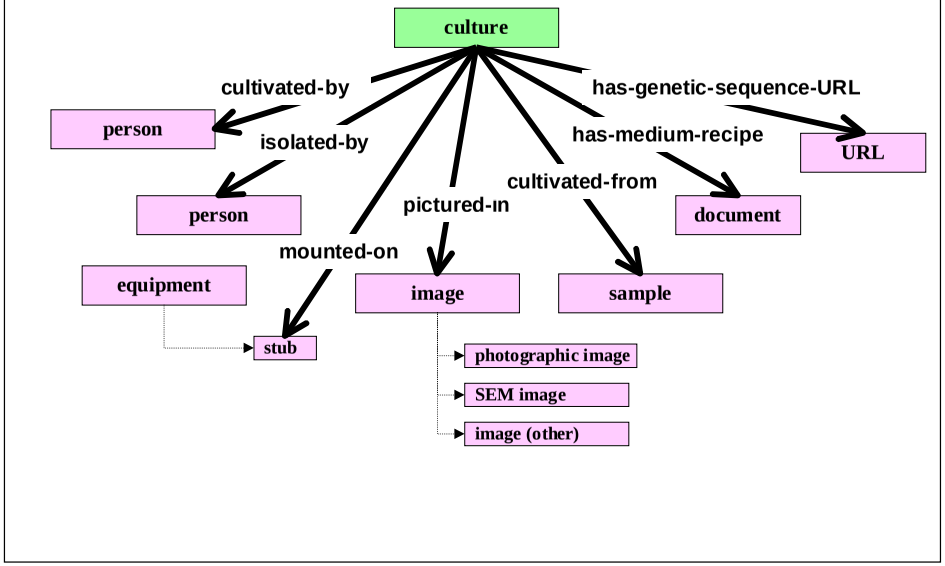
Information Resources ("nodes")

- Describe various types of project-specific information: People, Places, Events, Devices, Measurements
  (e.g., field sites, labs, trips, samples, images, documents, instruments etc.)
- Contain metadata (categorical, text, or numeric)
- Can have "attached" files (e.g., images, documents, datasets)

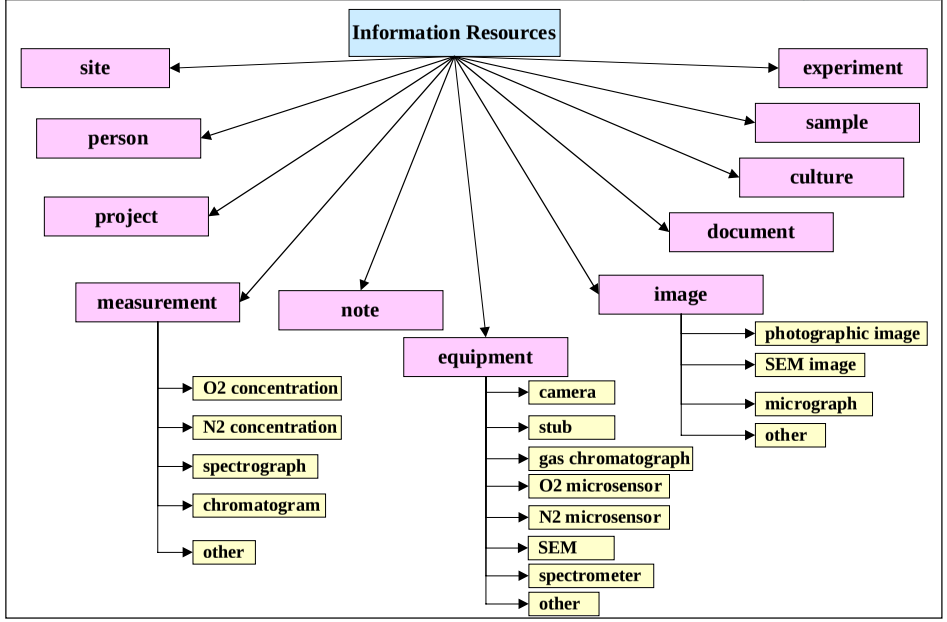
Examples of Information Resources:



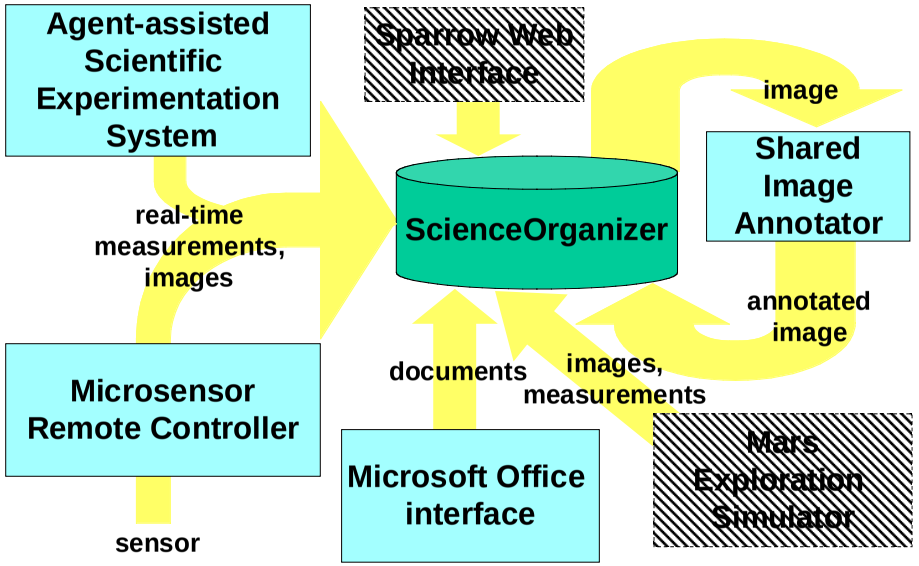
Sanctioned Links between "culture" and other types of resources



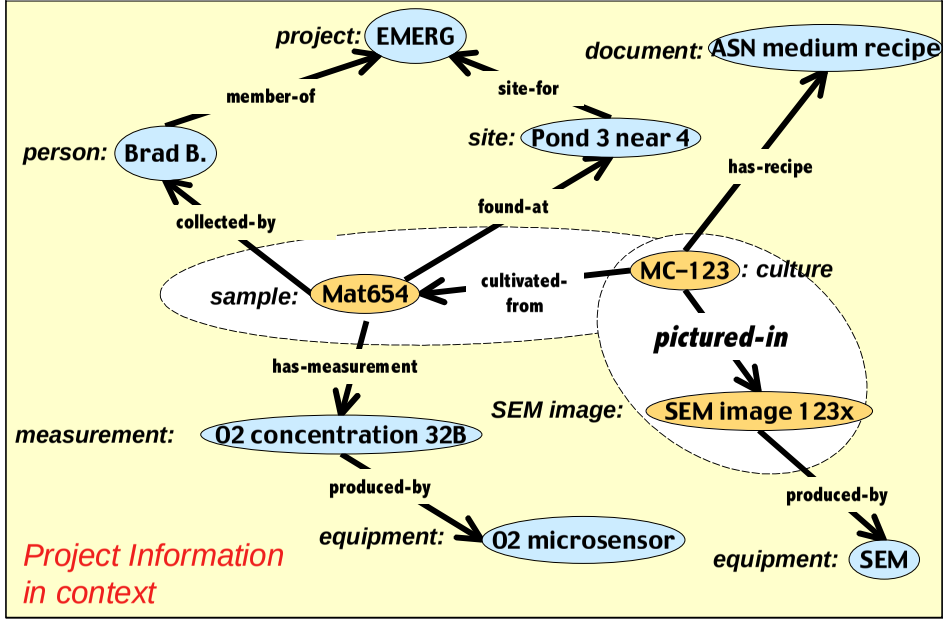
Types of ScienceOrganizer "Information Resources" (partial)



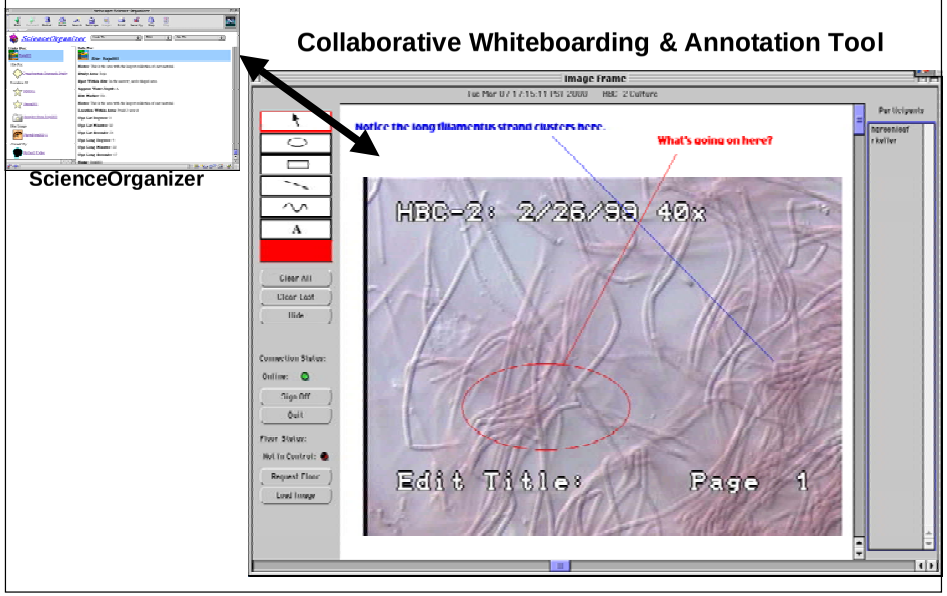
Interfacing Software



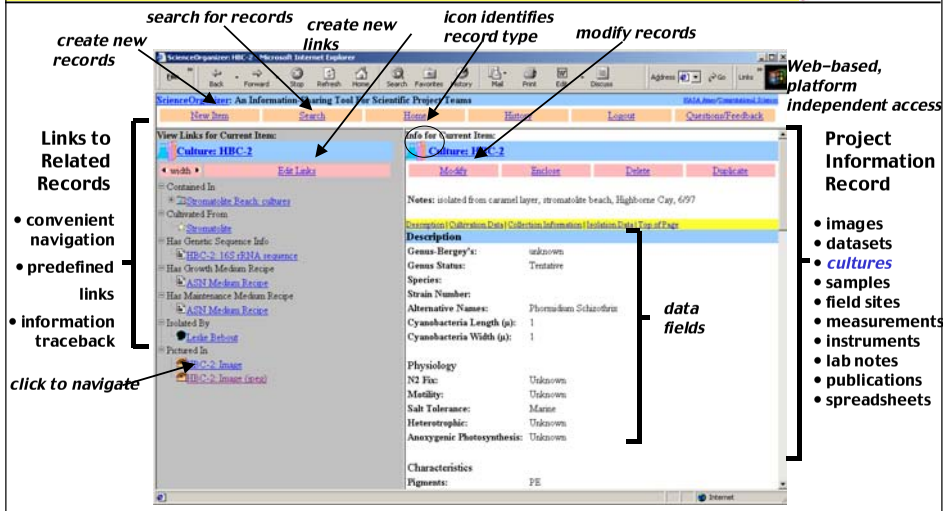
Evolving Web of Project Information



Shared Image Annotator



ScienceOrganizer Browser



ScienceOrganizer Customers

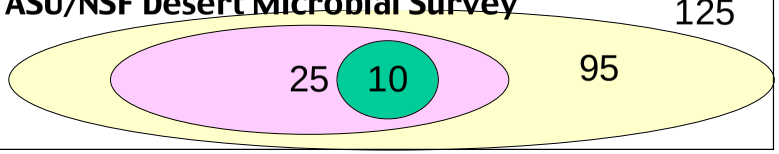
more mature

less mature

Pilot Users:

- ARC Microbial Ecosystems Group
- NAI Ecogenomics Focus Group
- ECS Mishap Investigation
- ARC Electron Microscopy Lab
- JSC Astrobiology Institute for the Study of Biomarkers
- NIH/NASA Malaria Control Study (via Fundamental Biology Program)
- Mars simulation and analog studies (via Mobile Agents Project)
- ASU/NSF Desert Microbial Survey

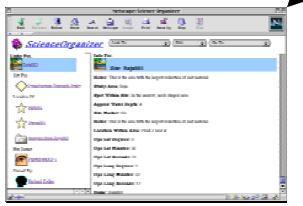
Users
- Frequent
- Moderate
- Infrequent

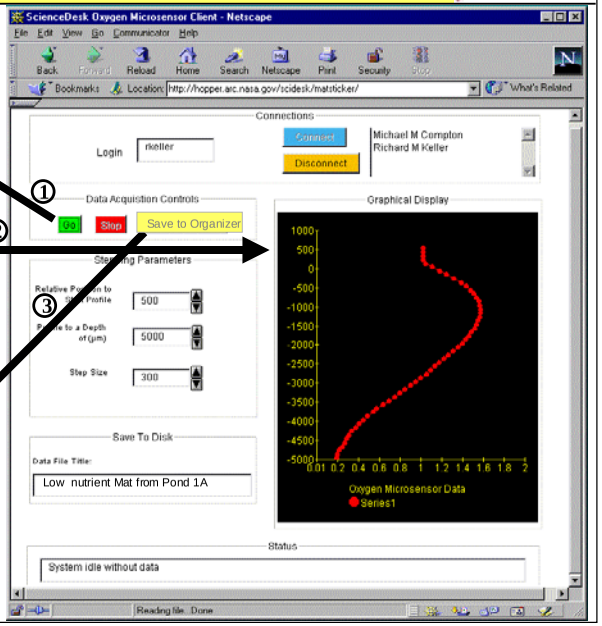


Microsensor Remote Controller

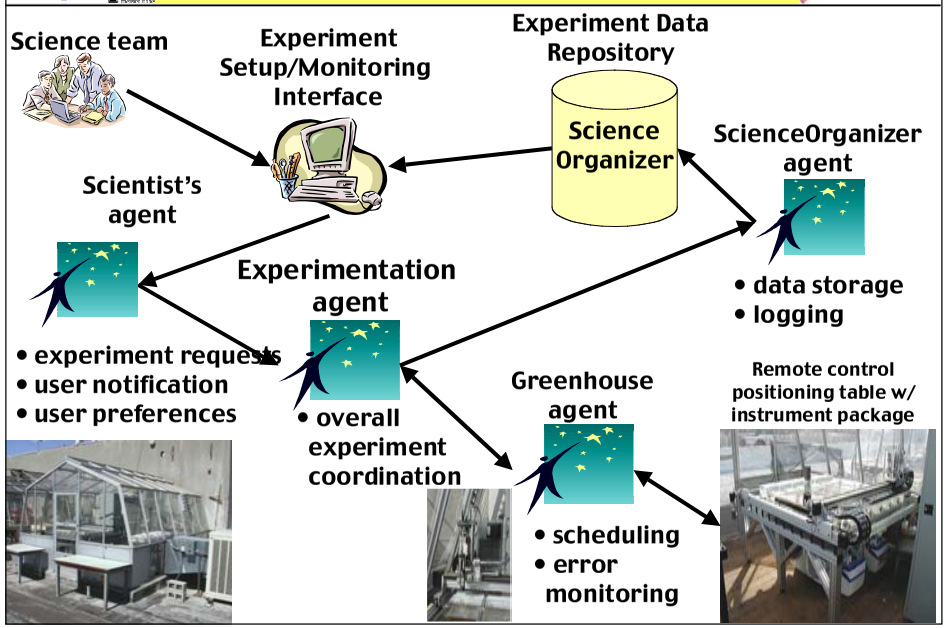
O_2 Microsensor & positioning table

ScienceOrganizer

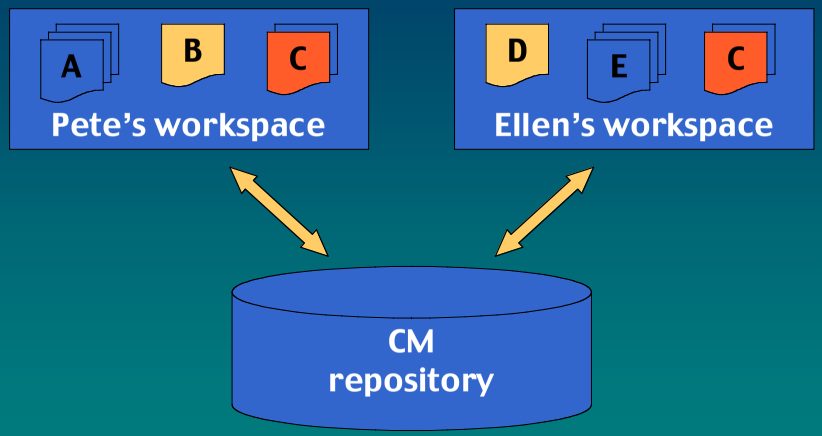




Agent-assisted Scientific Experimentation System



Direct Conflicts

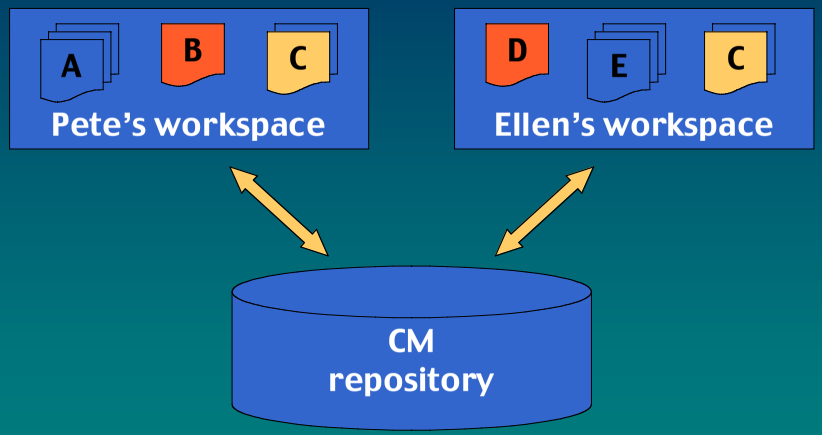


Overlapping changes to the same artifact

Palantír: Increasing Awareness among Distributed CM Workspaces

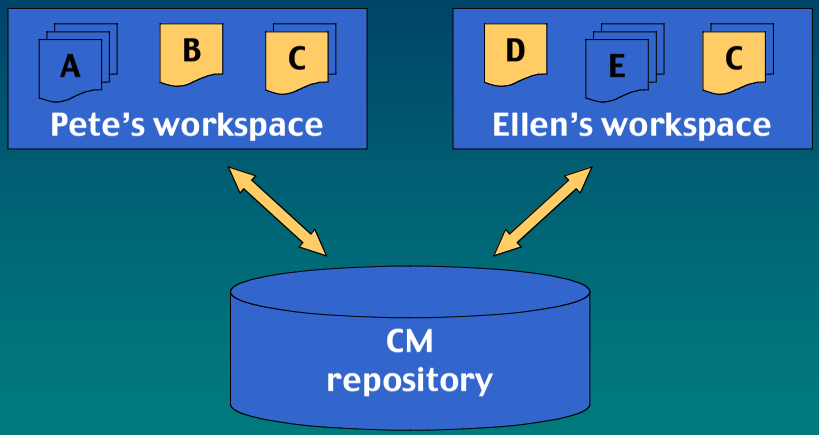
André van der Hoek, Anita Sarma
Institute for Software Research
University of California, Irvine
{andre,asarma}@ics.uci.edu

Indirect Conflicts



Changes to one artifact modifying the behavior of another artifact

A Typical Development Scenario



Key Observation

- A CM workspace in reality provides two kinds of isolation:
  - Good isolation
    - Hides actual changes to artifacts
  - Bad isolation
    - Hides knowledge of what artifacts other developers are changing

Traditional CM Approaches

- Pessimistic
  - An artifact can be changed by only one person at any one time
  - Limited in not allowing any parallel work
- Optimistic
  - An artifact can be changed by many persons at the same time
  - Limited in leading to merge problems that need to be resolved manually

Approach

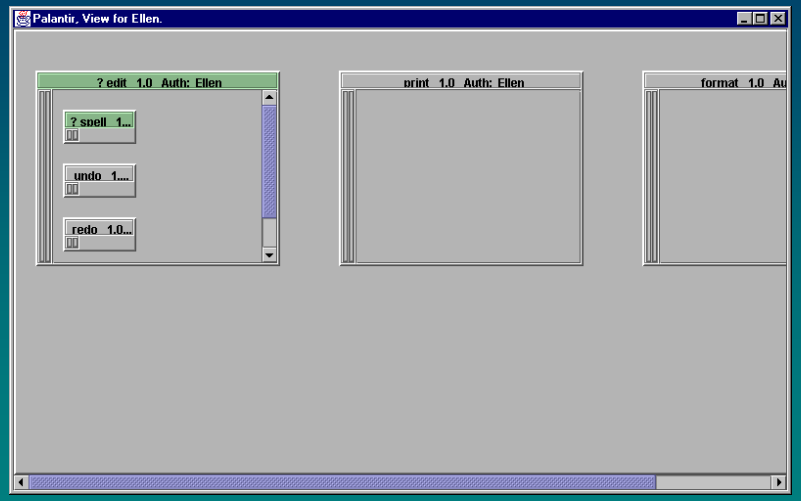
- Continuous workspace awareness
  - Which artifacts are being changed by whom?
  - What is the *severity* of the changes? (amount/size of change being made)
  - What is the *impact* of the changes? (effect of changes on one's current work)
- Such awareness has the potential to significantly reduce the number of direct and indirect conflicts

Traditional CM Approaches

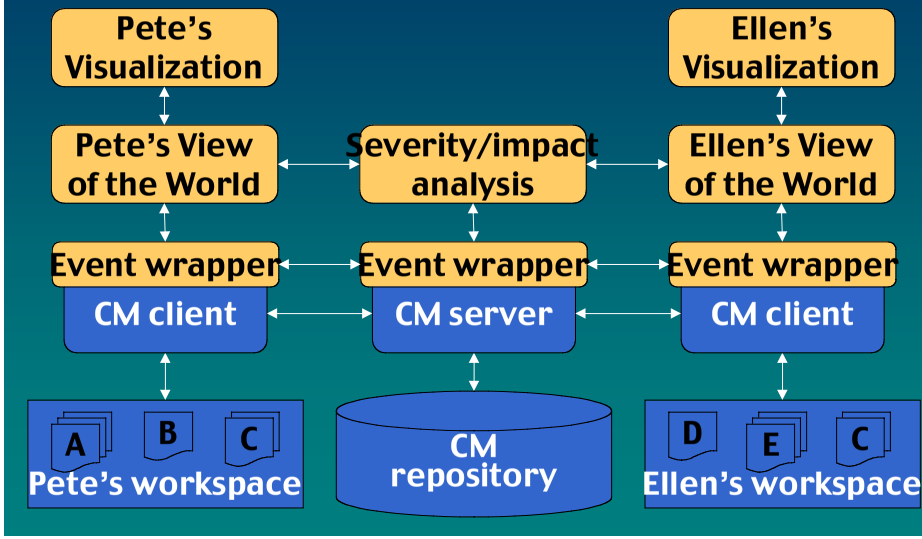
- Pessimistic
  - An artifact can be changed by only one person at any one time
  - Limited in not allowing any parallel work
- Optimistic
  - An artifact can be changed by many persons at the same time
  - Limited in leading to merge problems that need to be resolved manually

Neither solution addresses direct and indirect conflicts very well, especially in a distributed and decentralized setting

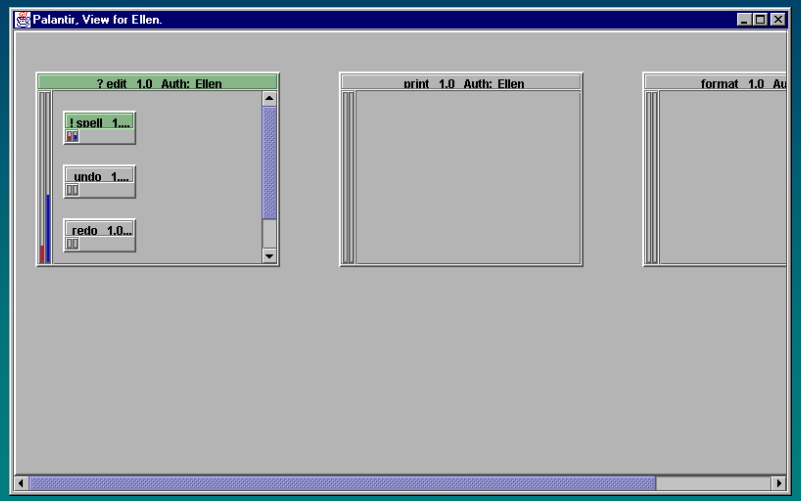
Making Changes in the Workspace



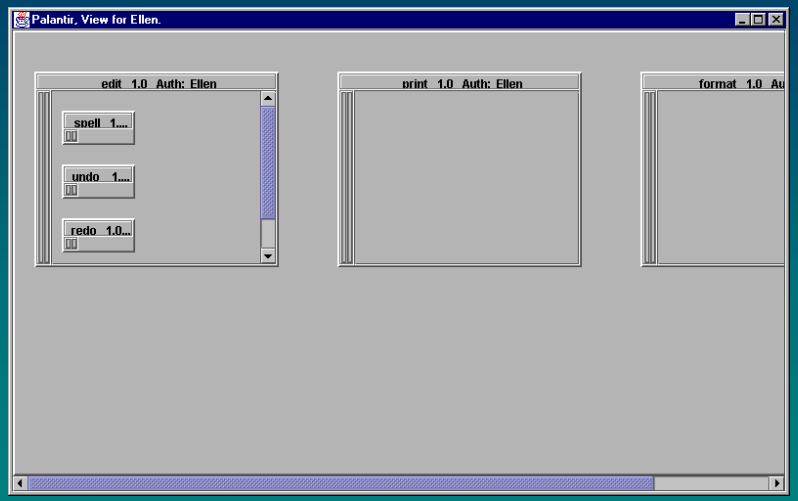
Palantír Architecture



Committing Changes



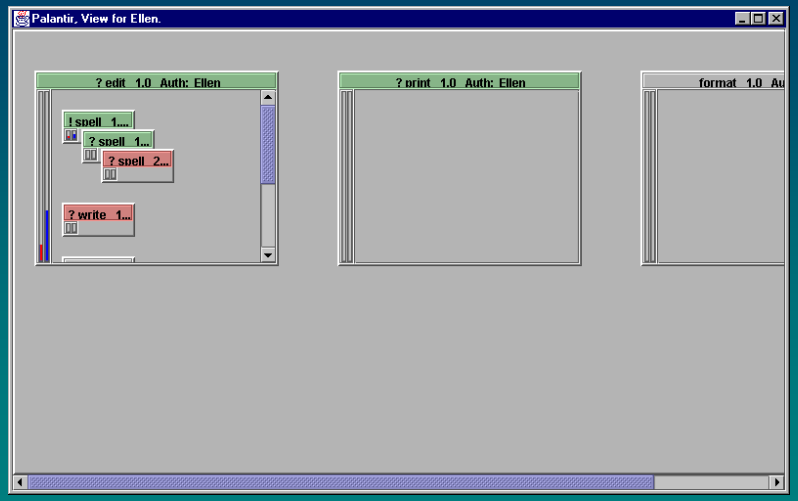
Populating a Workspace



Visualization Features

- Different views with different trade-offs
  - Amount of information versus level of intrusiveness
  - Scroll-bar, tabular, fully graphical
- Configurable
  - Selection of relevant developers, events, timeframes
- Scalable
  - Internal data structure versus actual visualization
  - Pair-wise workspaces
  - Sorting per severity or change impact
- Extensive metadata

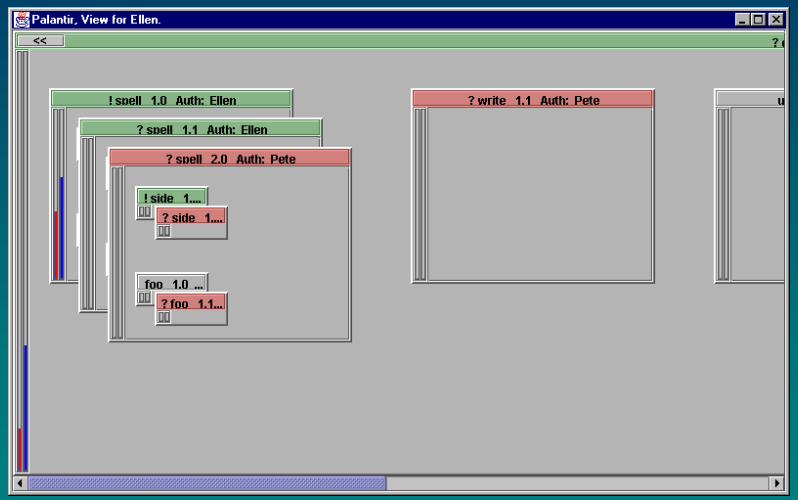
More Changes (by Other Developers)



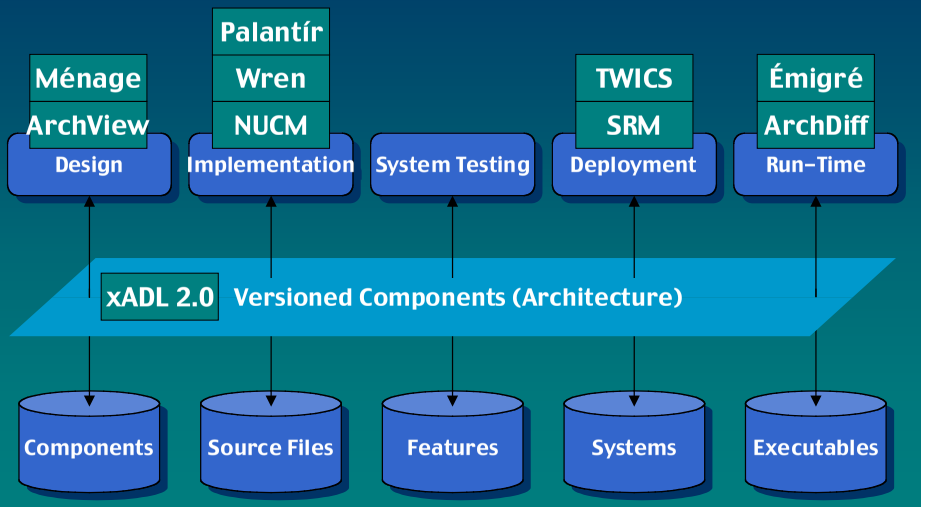
Severity Analysis

- Amount (size) of change being made
- Proposed algorithms
  - Number of files
    - Simple, but inaccurate
  - Lines of code
    - Simple, but inaccurate
  - Token based difference
    - Measures structural changes, but language dependent
  - Abstract syntax tree
    - Very detailed analyses, but likely too expensive (and language dependent)
- Current work in progress

Etc., Etc., Etc...



Research Projects



Impact Analysis

- Effect of changes on one's current work
- Proposed algorithms
  - Overlapping number of files
    - Simple, but inaccurate
  - Overlapping lines of code
    - Simple, but inaccurate
  - Changed interfaces
    - Potentially accurate and effective, but language dependent
  - Dependency analysis
    - Very precise, semantic results, but complex (and language dependent)
- Current work in progress

Conclusions

- Palantír is a prototype that...
  - ...brings awareness to distributed CM workspaces
  - ...shows pair-wise conflict
  - ...provides severity and impact analyses
- Palantír is independent of the type of CM system used
- Use of Palantír results in fewer direct and indirect conflicts
  - Case study to be planned in near future
- Future work
  - Integrate with different CM systems
  - Implement severity and impact analysis algorithms for both atomic and compound artifacts
  - Develop additional visualizations

Hypothesis and Proposal

- Hypothesis: an analysis of historical modification data can identify and classify problematic, high-maintenance software regions.
  - These regions can be described as “instabilities”.
  - Such knowledge can direct software redesign efforts.
- Proposal: IVA, a tool to visualize and analyze software instabilities.
  - The visualization can direct focused analysis.

IVA: Visualizing Software Instability

Jennifer Bevan
University of California, Santa Cruz
Jbevan@soe.ucsc.edu

Related Research

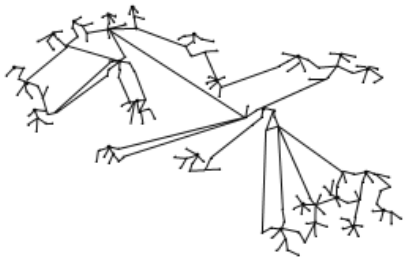
- Static software analysis…
  - Uses dependence graphs of a single revision to generate code metrics (cohesion, coupling, complexity) or conduct change impact analyses.
- Software evolution either…
  - Analyzes software modification data to create process-level metrics and models of evolution.
  - Attempts to automatically evolve software.

Problem: Software Decay

- Software decays as a result of incompatibilites between the operating environment and the implemented artifact.
  - Failure to meet requirements, specify accurate requirements, or anticipate changes in requirements.
- The existing software architecture can hinder the effectiveness of the maintenance process.
  - “golden handcuffs”, intransigent code.

Instability Visualization (1 of 3)

- Dependence graph nodes positioned using heirarchical relationship.
- Causes spatial clustering of related nodes:
  - Package, class, method
  - Directory, file, function

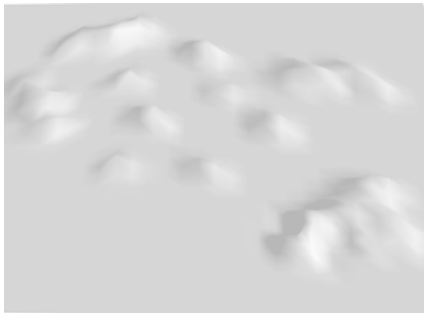


IVA Is Different Because…

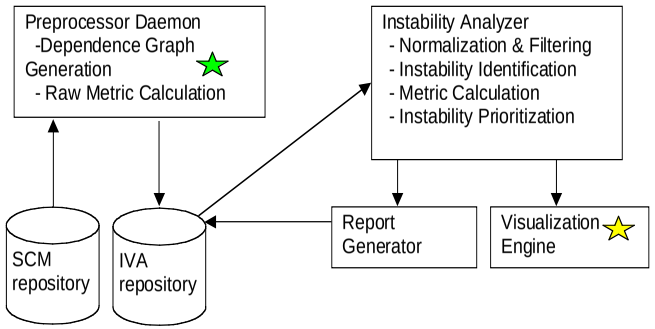
- IVA distinguishes between dependence-related changes and changes made during the same “commit”.
- IVA does not require advanced change management data for basic functionality
  - Only requires when, where, what, but not why.
- User controls IVA filtering and aggregating of change data.
  - Different users are interested in different things.

Instability Visualization (2 of 3)

- Surface map generated from dependence graph layout.
- Retains global context of data (code location, etc.)
- Hides edges, reduces clutter.



IVA Architecture

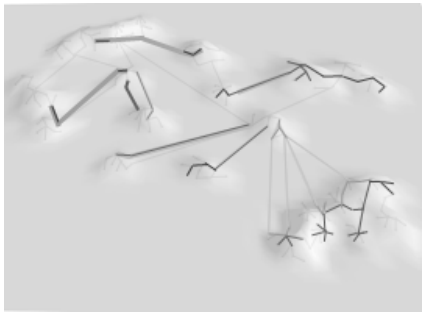


Conclusion

- IVA will leverage the data stored in change control systems (CVS as a minimum) by identifying and classifying historical change patterns.
- A proof-of-concept IVA is under construction
  - Will handle Java source code in Subversion repository.
  - Will provide additional visualizations for in-depth exploration of specific instability regions.

Instability Visualization (3 of 3)

- Classified instability regions are overlaid on the surface map.
- Instabilities follow edges of underlying dependence graph.
- Color and width denote user-controllable metrics; distance denotes span of coupling.



Questions?

- The work completed to date was funded by a 2001 USENIX Student Research Grant.
- See http://www.cse.ucsc.edu/~jbevan for IVA progress and status updates.
- Email jbevan@cse.ucsc.edu with future questions.

Use In Collaborative Development

- IVA can analyze and provide feedback on a given implementation of collaborative development.
  - Does task breakdown force contention?
    - Coloration based on number of different committers.
  - Does system architecture force contention?
    - High severity and number of different committers.
  - User can control visualization by directing color, line width, or aggregation algorithms.

Examples of Software Concerns

- **Security**
  - Always call the security check before allowing database access
- **Accounting**
  - Always debit the user's account on each access to a service of objects in the class…
- **Synchronization**
  - Don't let multiple users call any of methods *f*, *g*, or *h* on a single object simultaneously
  - The effects of these actions should be transactional
- **Quality of service**
  - Queue up the waiting calls handling them by priority
- **Reliability**
  - Provide replicants of this object
- **Performance enhancements**
  - Cache the results of calls to elements in this class
  - Display routines should show the results of changes, except display routines called in the scope of other display routines should buffer their changes for display all at once

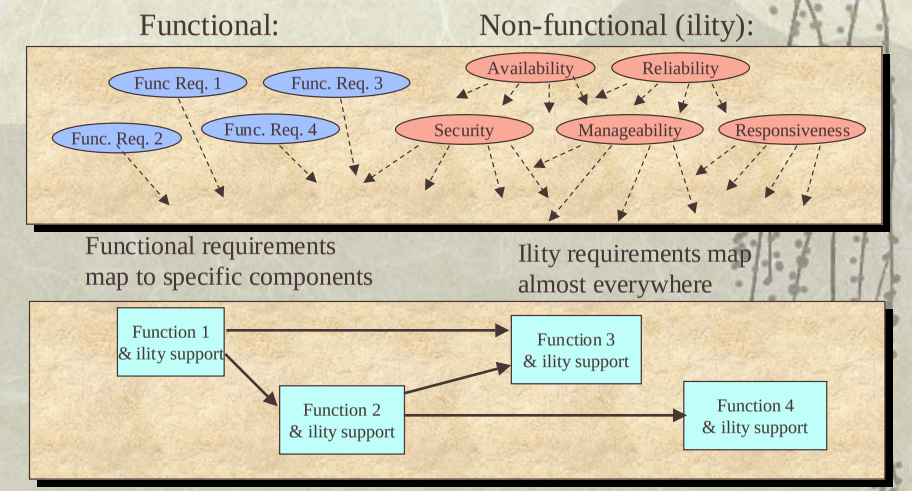
Source-Code Instrumentation and Quantification of Events

Robert E. Filman
RIACS
rfilman@mail.arc.nasa.gov

Klaus Havelund
Kestrel Technology
havelund@email.arc.nasa.gov

NASA Ames Research Center
Moffett Field, CA 94035 U.S.A.

Functional and Non-Functional Requirements



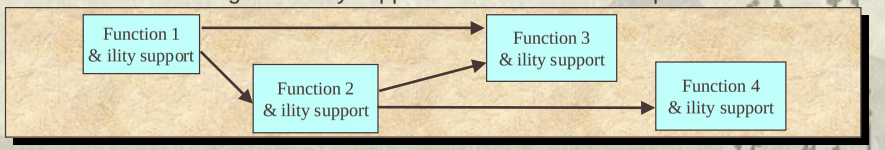
Separation of Concerns

- **A fundamental engineering principle is that of *separation of concerns***
  - Realizing different system concepts as separate, weakly linked elements
  - Distribution of expertise
- **Separation of concerns promises better**
  - Maintainability
  - Evolvability
  - Reusability
  - Adaptivity
- **Concerns cross-cut**
  - Apply to different modules in a variety of places
- **Concerns must be composed to build running systems**

In conventional programming, the code for different concerns often becomes mixed-together (*tangled*)

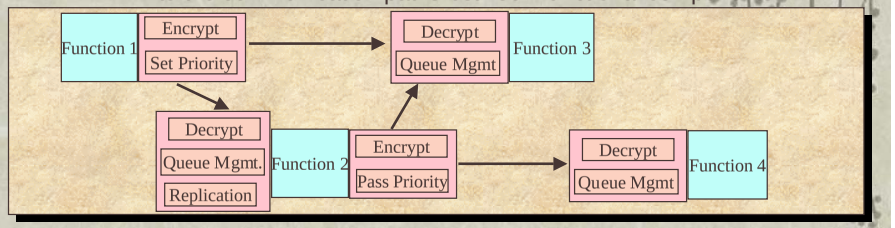
Architecture with Services in Component Communications

Traditional designs mix ility support within functional components:



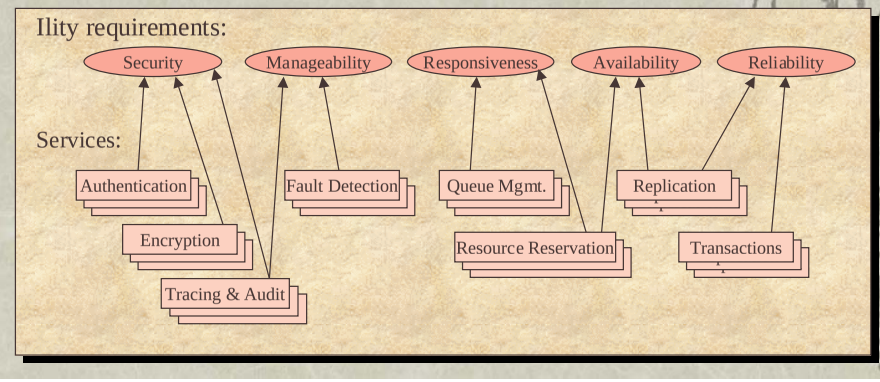
Key insight: Service functionality can be separated from functional logic by inserting the services into the communications between functional components.

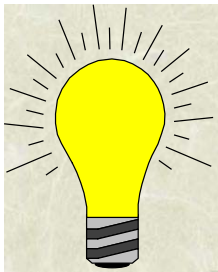
OIF is demonstrating frameworks that insert services into the communication paths between functional components.



Services and Ilities

- Ility requirements are implemented by combinations of service algorithms.
- Supporting ilities involves a complex selection from sets of alternative service algorithms.
- The services must be invoked pervasively throughout the system.





Key OIF Ideas

- Injecting behavior on the communicate paths between components
  - Injectors are discrete, uniform objects
  - Injectors are by object/method
  - Injectors are dynamically configurable
- Annotated communications allow injected services to pass parameters to service peers (e.g., message priority, user-id, tracing status)
- Thread contexts preserve annotations through calls
- Pragma: High-level specification language for describing desired injections (Pragma)

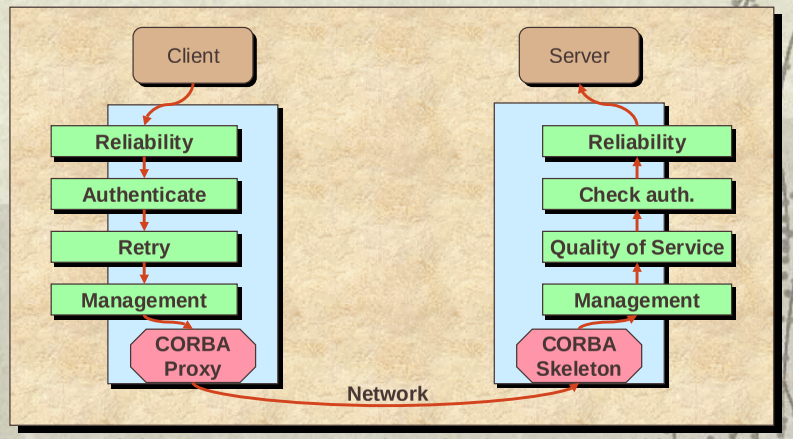
Object Infrastructure Framework Research Hypothesis

- Ilities can be achieved by inserting services into the communication path between functional components
  - On both sides of the communication
- Frameworks that automate service insertion can systematically achieve non-functional requirements
  - Object Infrastructure Framework (OIF)

Aspect-Oriented Programming (AOP)

- OIF is an Aspect-Oriented Programming system
- Software engineering technology for separately expressing systematic properties while nevertheless producing running systems that embody these properties
- Need to express
  - Base program
  - Separate concerns
  - How the separate concerns map to the base program
    - Or, if you prefer, just a jumble of program elements that must be combined.

Configurable Proxies



- OIF's injectors can operate in pairs (e.g., encrypt/decrypt; request authentication/authenticate) or singly (e.g., retry on failure; log results)
- Configuration is by proxy/method instance
- Configuration is dynamic

Quantification and Implicit Invocation

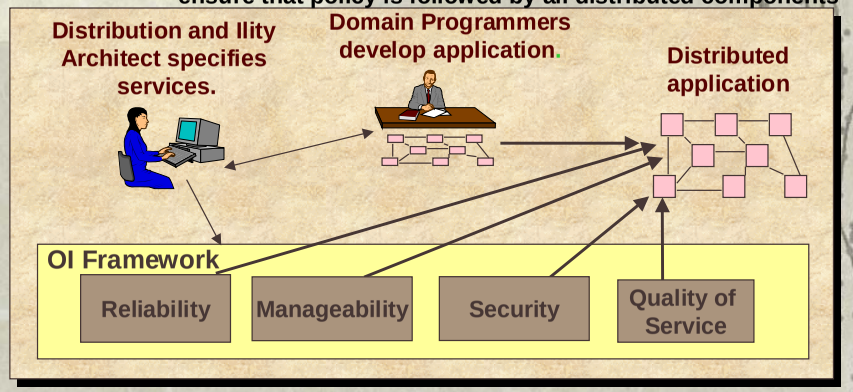
- Programmatic essence of AOP is to be able to state universally quantified statements about (conventional, linear) programs
- And have the effect of these statements realized in programs that don't contain explicit notation applying these quantified statements

In programs P, whenever condition C arises, perform action A.

- Dimensions of concern for the designer and implementer of an AOP system:
  - Quantification: What kinds of conditions C can be specified.
  - Interface: What is the interface of the actions A. That is, how do they interact with base programs and each other.
  - Weaving: How will the system arrange to intermix the execution of the base actions of P with the actions A.

OIF Process

- Map organizational *policies* to implementation
  - E.g.,
    - define a security policy
    - ensure that policy is followed by all distributed components



Extreme Experiment

- The extreme of expressiveness in quantification is to be able to quantify over all the history of events in a program execution
- Events are with respect to the abstract interpreter of a language
- Unfortunately, language definitions don't define their abstract interpreters.

Choices in Developing AOP Languages

- What quantified statements are allowed
  - Join points
    - Method calls
    - Field access
    - Abstract syntax
    - Control flow
  - Scope of quantification
    - Subclasses
    - By name
    - Lexical structure
- Syntax for expressing application
- Interaction among aspects and base code
  - Visibility
  - Ordering
  - Conflict resolution
- Implementation mechanism
  - Compiler
  - Byte-code manipulation
  - Dynamic wrapping
  - Frameworks
  - Meta-programming
  - Program transformation

Events and Event Loci

Event	Syntactic locus
Accessing the value of a variable or field	References to that variable
Modifying the value of a variable or field	Assignments to that variable
Invoking a subprogram	Subprogram calls
Cycling through a loop	Loop statements
Branching on a conditional	The conditional statement
Initializing an instance	The constructors for that object
Throwing an exception	Throw statements
Catching an exception	Catch statements
Waiting on a lock	Wait and synchronize statements

Static and Dynamic Quantification

- In earlier work, we distinguished between
  - Static quantification: discernable from the syntactic structure of the specimen program
    - E.g, calls
  - Dynamic quantification: matching events that happen in the course of program execution.
    - E.g., cflow
- Currently exploring the hypothesis that (almost) all interesting "events" are dynamic, and that static quantification merely refers to those events that can be simply inferred from the static structure of the program.
  - Shadows in the code
  - Exception: program structural changes

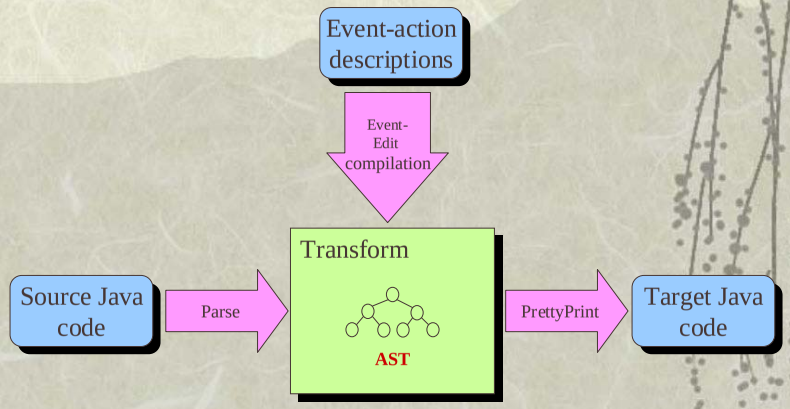
Transformational Alternatives

- For Java, can transform at
  - The source-code level
  - The byte-code level

More Events and Loci

Event	Syntactic locus
Resuming after a lock wait	Other's notify and end of synchronizations
Testing a predicate on several fields	Every modification of any of those fields
Changing a value on the path to another	Control and data flow analysis over statements (slices)
Swapping the running thread	Not reliably accessible, but atomization may be possible
Being below on the stack	Subprogram calls
Freeing storage	Not reliably accessible, but can try using built-in primitives
Throwing an error	Not reliably accessible; could happen anywhere

Architectural View



Research regime

- Define a language of events and actions on those events.
- Determine how each event is reflected (or can be made visible) in source code.
- Create a system to transform programs with respect to these events and actions.

Detect Multi-threading Problems

- Deadlocks: **Observe in what order locks are taken and released and infer potential deadlocks from cycles.**
- Data Races: **Observe what locks threads own when they access variables and infer potential data races from empty overlaps.**

Applications

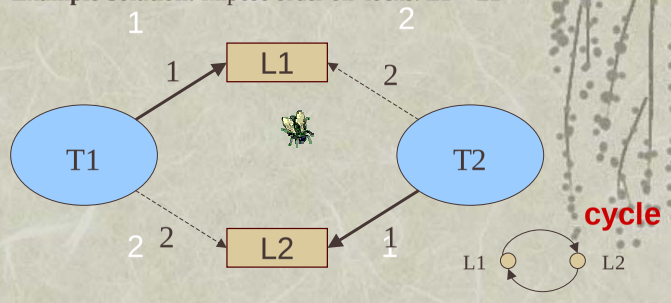
- **Applying AOP to debugging and validating concurrent programs.**
- **Applying AOP to monitor programs during operation, so that actions can be initiated in case bad things happen.**
- **Applying AOP as a general programming paradigm.**

Deadlocks

A deadlock can occur when threads access and lock shared resources, and lock these in different order.

Example Solution: Impose order on locks: L1 < L2

Problem:
T1 locks L1 first
T2 locks L2 first



Program Debugging

- **Detect multi-threading problems caused by access to shared resources by competing threads.**
- **Validate trace executions against user requirements.**
- **Validate multithreaded programs by exploring schedule interleavings.**

Data Races

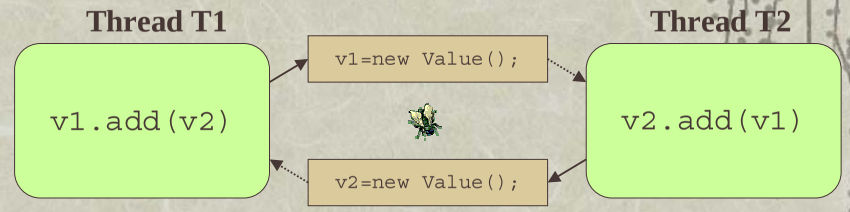
A data race occurs when two threads
- Access a shared variable,
- At least one access is a write, and
- No mechanism is used to prevent simultaneous access.

Example Solutions: monitors, semaphores, …

Result after both updates : 2 … or maybe 1

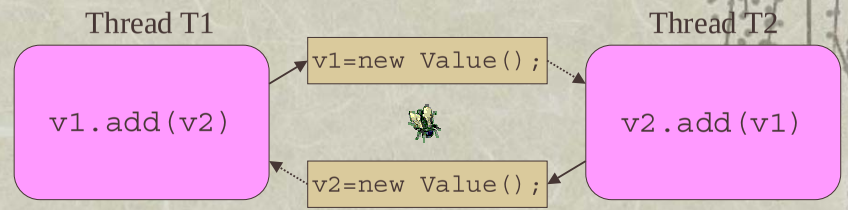
Java Program with Deadlock

```
class Value{
  int    x = 1;
  synchronized void add(Value v){x = x + v.get();}
  synchronized int  get(){return x;}
}
```



Java Program with Datarace

```
class Value{
  int x = 1;
  void add(Value v){x = x + v.get();}
  int get(){return x;}
}
```



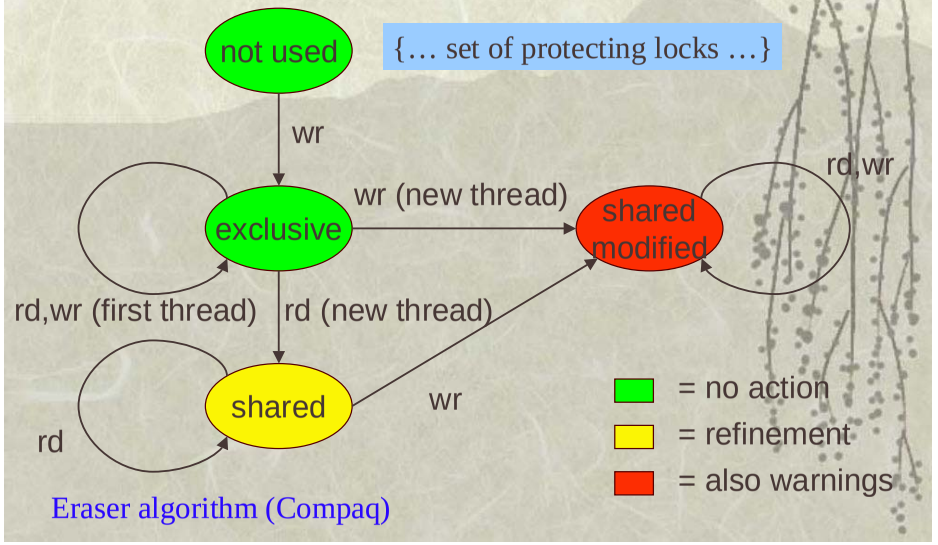
Applying AOP

```
aspect DeadlockDetection{
  when synchronize(obj){
    Thread curr = Thread.currentThread();
    Set locks = Threads.getLocks(curr);
    Graph.addEdges(locks,obj);
    Graph.findCycles();
    Threads.addLock(curr,obj);
  }
  when endof synchronize(obj){
    Threads.remove(curr,obj);
  }
}
```

Validating Execution Traces Against User Requirements

```
aspect CheckRequirements{
  when(CLOSED and not previously DO_CLOSE){
    Report("System closed by itself");
  }
  whennot(DO_CLOSE implies
            eventually(20)CLOSED){
    Report("System did not close");
    CloseSystem();  // repair
  }
}
```

For Each Variable: A Lockset and a Statemachine



Eraser algorithm (Compaq)

Explore Scheduling

- Simple example: assume that all variable accesses are protected with locks.
- Insert a call of a randomized yield statement in front of all synchronization statements and calls of synchronized methods.
- This will cause the scheduler to randomly make a context switch whenever a lock is taken. This may be used, for example, to reveal deadlocks.

Applying AOP

```
aspect DataraceDetection{
  when synchronize(obj){
    Thread curr = Thread.currentThread();
    Threads.addLock(curr,obj);
  }
  when endof synchronize(obj){
    Thread curr = Thread.currentThread();
    Threads.remove(curr,obj);
  }
  when accessto(var,isWrite){
    Thread curr = Thread.currentThread();
    Statemachine.update(curr,var,isWrite);
    Statemachine.checkEmptyness(var);
  }
}
```

Related work

- De Volder et al. metaprogramming
- AOP through program transformation
  - Colcumbet, Fradet and Sudholt
  - Schonger et al. XML transformation
  - Skipper ku
- Nelson et al. concern-level foundational composition operators: correspondence, behavioral semantics and binding
- Walker and Murphy on events as join points

Concluding remarks

- Presented
  - AOP
  - OIF, an AOP system
  - Event-based quantified approach to AOP
    - Developing an environment for experimenting with AOP languages (DSL for AOP)

approach

- visual approaches
  - cognitive -> perceptual
  - focus on artifacts rather than processes
- theoretical background
  - the experience of computation
    - making computation "present"
  - embodied interaction
    - interaction as an embodied practice
    - directness and engagement
- two projects: seesoft and vavoom

Visualization of Software and Activity

Paul Dourish
Institute for Software Research
UC Irvine
jpd@ics.uci.edu

NASA ARC, August 2002

seesoft

- cscw research focus on awareness
  - passive understanding of the activity of others
- trying to understand "awareness in the large"
  - large-scale, distributed software development
  - in large projects, the codebase is the artifact
  - awareness of changes in the artifact
  - awareness of the actions of others
- seesoft
  - adopted from work of eick, wills, et al.
  - view the activity in cvs repositories

what we already know

- from studies of software development
  - complexity of task and interaction
  - the impact of distance
  - complexity of interdependence
- from studies of collaboration
  - formal and informal
  - the role of awareness
    - qualitative understandings of the actions of others
    - provides a context for your own actions

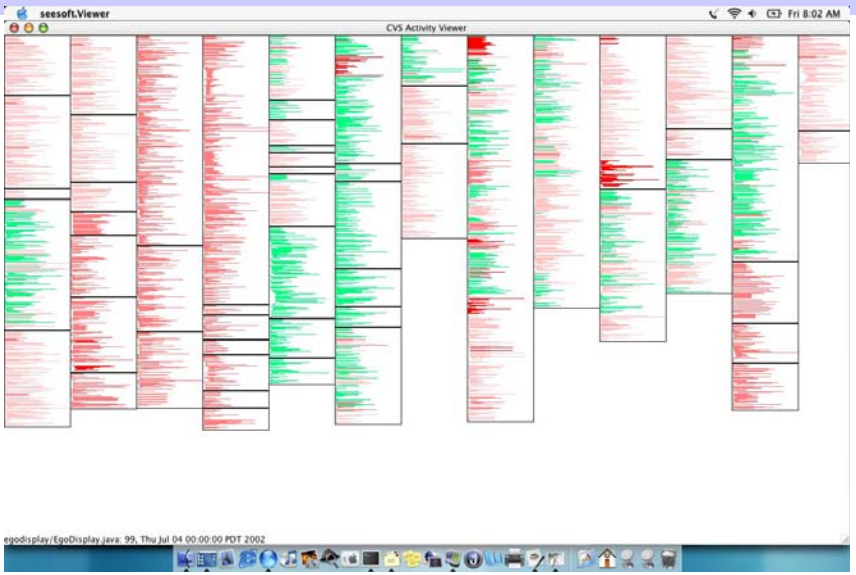
seesoft

seesoft

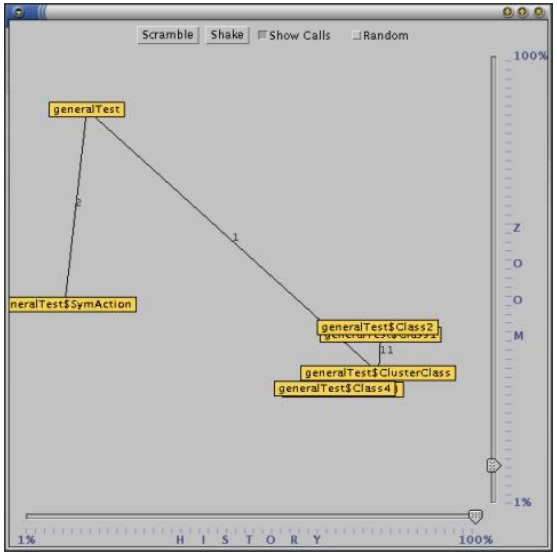
vavoom

- interface abstractions hide mechanism
  - "folders" are folders whether local or networked
- but mechanism is how we understand the world
  - understandings of cause and effect
  - temporal dynamic coupling
  - the temporal organization of social action
- manifesting computation
  - how to give people a picture of what's happening?

seesoft

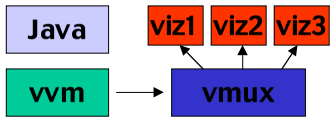


vavoom

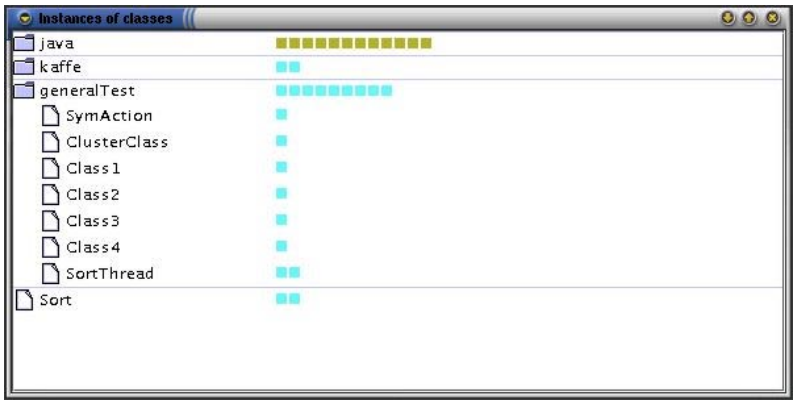


vavoom

- initial exploration: vavoom
  - focus on novice programmers
    - we understand the models in terms of which to explain
    - vested interest in finding out what's going on
    - there's plenty of them lying around
  - vavoom is the visual virtual machine
    - visualize java program execution
    - dynamic, real-time visualizations
    - unmodified class files



vavoom

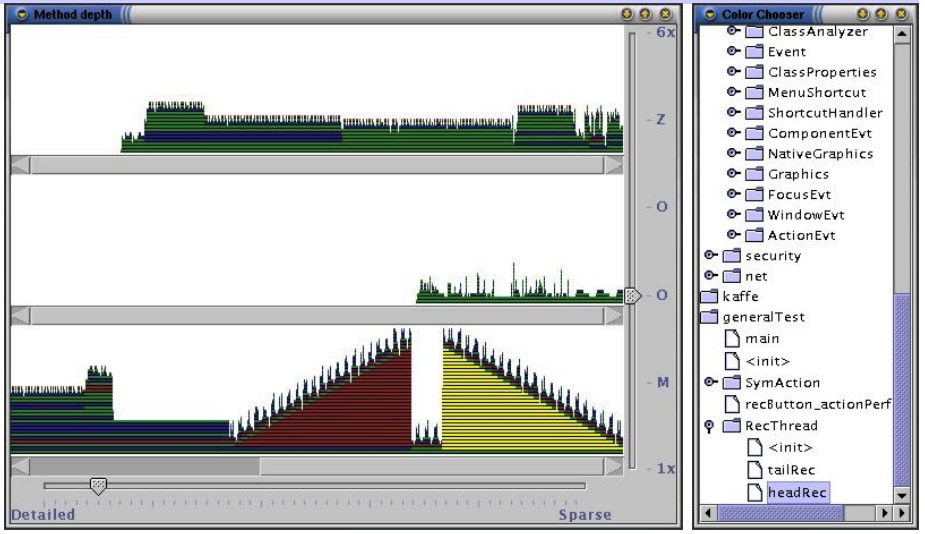


vavoom

conclusions

- system support for informal interaction
  - non process-centric collaboration
  - qualitative understandings of
    - software behavior
    - collaborative activity
- exploiting software as artifact
  - software as primary coordinative resource
  - directness and malleability
- open questions
  - integration with practice
  - balance between translucence and distraction
  - demonstration of theoretical approach

vavoom



vavoom

- vavoom is an early technical exploration
  - the focus on programmers is a hack
    - but convenient… as long as they have the right programs
  - a more interesting strategy is to:
    - focus on end users
    - focus on more abstract visualisations
    - focus on more specialised tasks
      - currently exploring network security

Research Questions

- In practice, does the order of first determining project choices, making project selection and then performing requirements analysis hold?
- If not, what are possible procedural relationships between project choice construction, project selection and requirements analysis?
- How are they similar or different to current requirements analysis views?

Exploring the Relationship between Project Selection and Requirements Analysis

Mark Bergman, Gloria Mark
Information and Computer Science Dept.
University of California, Irvine
mbergman@ics.uci.edu, gmark@ics.uci.edu

Research Methods

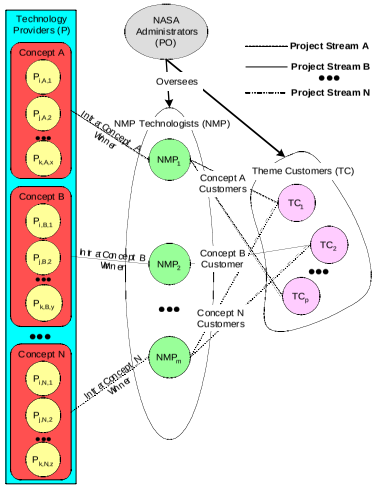
- Project Selection and Requirements Analysis have been studied individually, but not together
  - Project Selection has been examined empirically
  - Almost no *in situ* Requirements Analysis studies
- Apply Ethnographic Methods to study initial project formation *in situ*
  - 5 months (2-3 times weekly) of on site participant observation
  - 46 individual semi-structured interviews and 34 semi-formal and formal group meetings
  - 5 detailed technical presentations
  - Hundreds of related documents

Initial Project Formation

- First, Project Selection
  - Determine project choices
  - Choose a project to fund and develop
- Then, Requirements Analysis
  - Determining stakeholders' wants, needs, and constraints for a project
  - Requirements Analysis traditionally follows Project Selection
- How does Project Selection relate to Requirements Analysis?
  - Project Selection decisions frame subsequent Requirements Analysis

Roles and Project Selection

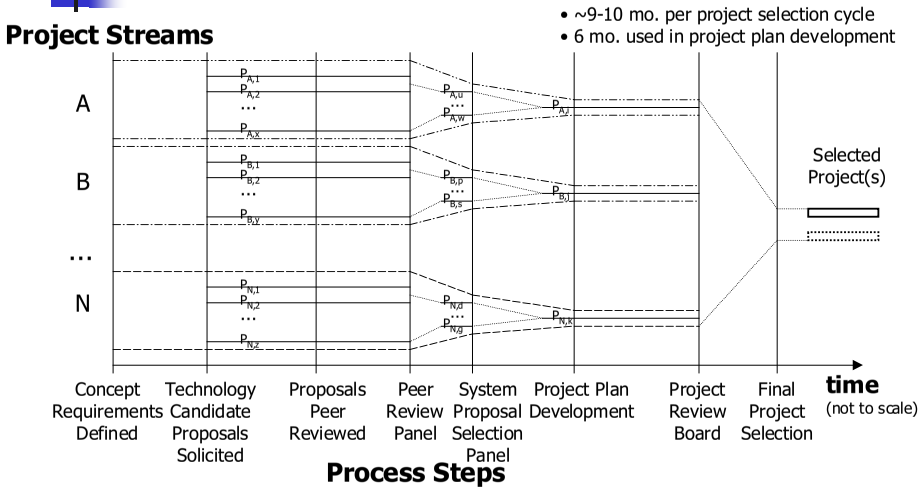
Process Role	Lab Role
PO: Process Owners/ Principals	NASA Administrators
TC_p: Theme Customers	One of p Mission Themes
P_i: Technology Providers	One of i Technology Providers
NMP_m: Process Actors/Agents	One of m NMP Technologists
Concept N	One of N Competing General System Candidates
$P_{i,N,x}$	One of x Competing Technologies for Concept N from Provider i
Project Stream N	One of N Candidate Project Plans, Project Plan N is for Concept N



The Field Site

- The New Millennium Program (NMP) at the Jet Propulsion Laboratory (JPL): A group in a NASA (National Aeronautics and Space Administration) research laboratory located in Southern California
- The NMP program's mission: Space flight validate new technologies that are deemed important to NASA's future science missions
  - This includes maturing new technologies (TRL 3 → TRL 7)
- NMP Selection Process: Choosing which new technologies to validate
- Each new technology candidate can become the basis of a new project – a validation mission
  - NMP selection process is a highly developed form of *in situ* initial project formation

Project Selection Process



Roles and Requirements

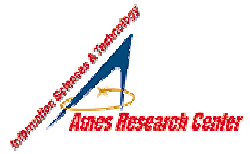
Lab Roles	Description	General Requirements Profile
NASA Administrators	NASA upper level decision makers with the authority to assign organizational resources to implement their decision	Wants, needs and constraints tend to be general, somewhat vague, and usually conflicting Want broadly applicable array of new technologies to become available for NASA wide science mission usage, while minimizing cost Constrained by budgetary and policy guidelines from the US Congress
Mission Themes	Planners, designers, scientists, builders, and managers of science mission space systems	Technically explicit and precise in their needs and constraints Want new technology that would lower future science mission system costs or enable experiments Constrained by tight budgets and project deadlines
Technology Providers	Builders of new aerospace related technologies	Have very precise constraints and usage guidelines while providing specific, semi-customizable technical functionality Want their technologies space flight validated, likely creating a long term revenue stream, while minimizing technology development costs Constrained by VAL award amounts and project deadlines
NMP Technologists	Assist and promote the technology and project selection process	Want new technologies to space flight validate Want to balance and satisfy the needs of the administrators and themes, while validating as many providers' technologies as possible Constrained by allotted project cycle budgets and given deadlines

Relationship Between Project Selection and Requirements Analysis

- Initial project streams (concepts) defined by Theme Customer's requirements
- Competition between technology candidates informs and refines project stream requirements
  - Early identification of *wanted* and *undesired* existing technological capability, costs and constraints
  - Technology selection frames project definition and tightens project requirements
- Competition between project streams also refines each project's requirements
  - Project level requirements are used in project selection decision, especially negative requirements

Multiple Parallel Competitive Requirements Analysis (MPCRA)

- Relationship between project choices, selection and requirements analysis is bidirectional, *not* unidirectional
  - Projects created and selected requirements
  - Requirements identified and framed projects, and informed project selection
- Found multiple parallel competitive requirements paths, not just one
  - Refutes traditional requirements analysis – single project path
  - Outcome could be multiple projects, as opposed to the traditional assumption of one
- Extensive, multi-step, well-documented, open, competitive process built consensus for final project selection



The Past and Future of Postdoc

A fifteen-minute overview of Postdoc and where document collaboration is heading.

Current Postdoc: Usage

- Developed at Ames for the New Millennium Program scientists in '94, re-written in '95
- free for NASA and NASA collaborators
- ~4500 active accounts (~2800 nasa.gov)
- 235 users logged in last week
- 60GB of data
- ~1,000 e-mail messages last week
- 1.5 FTE developers, 1.5 FTE user-support

Current Postdoc: Features

- document sharing using any web browser
- completely user-managed access control, groups, lists
- per-document access control
- automatic document conversion (MS Office -> PDF)
- integrated mailing lists
- (simple) locking/versioning
- easy-to-use, familiar user interface

Postdoc Future: Technologies

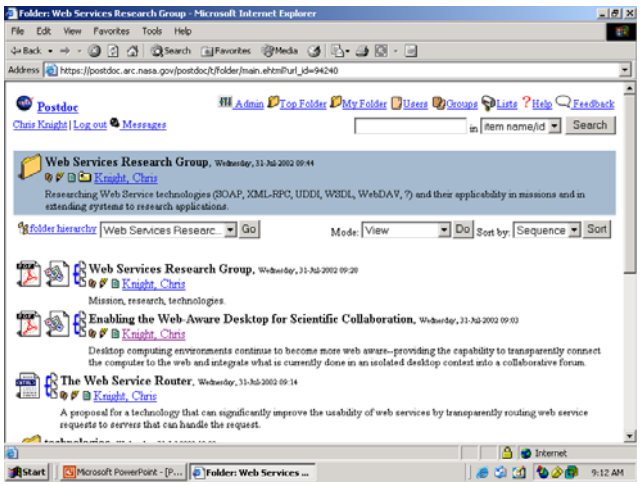
Integration with the desktop:

- WebDAV for "drag-and-drop" integration into modern desktops/applications (including ACL, DASL, DeltaV protocols)
- IMAP/NNTP for e-mail archive access
- multi-tier/modular architecture
- emphasis on standards-based programmatic access for application integration (Web Services)

Postdoc Future: Opportunities

- large user base
- large collection of a wide variety of documents
- long-term historical archive
- willing guinea pigs
- NASA-owned database, system, software
- hoping to work with mission(s)

Obligatory Screen Shot



The Setting:
Center TRACON Automation System (CTAS).

- A suite of automation tools developed at NASA/Ames designed to help air traffic controllers to manage air traffic flow at large airports.
- In 1991 it was chosen by the FAA as the future automation system for the terminal area.
- Since then it has been used in 6 different airports.

A Field Study of Collaborative Software Development Teams (Initial Results)

Cleidson de Souza[1,2]

John Penix[2]

Maarten Sierhuis[2]

[1]UC, Irvine Graduate Student and NASA/Ames Summer Intern

[2]NASA/Ames Research Center

The Setting:
Center TRACON Automation System (CTAS).

- CTAS is composed of 10 different tools.
- Source code:
  - C and C++. GUI's are being ported to Java.
  - 1,000,000 LOC .
- Development Team:
  - Number of developers: 31.
  - Two groups: V &V and Developers.
  - Work in processes, instead of tools

Overview

- The Setting: CTAS
- Methodology
- Initial Findings
- Future Work
- Conclusions

Initial Results

- Most important tools:
  - configuration management; and
  - bug tracking system.
- These tools provide shared repositories for source code and change requests.
- The CM and the bug tracking tool provide automation of some tasks like:
  - Version control, identification of releases, report generation, and so on.

Initial Results

- Developers adopt conventions to use these tools so that they users might cooperate effectively.
- Examples:
  - Naming conventions for creating branchs and views to work with the CM tool;
  - Priorities and severities of the bugs in the bug tracking tool.

Methodology

- Field Study
  - Five weeks in the field until now, four more weeks to go.
- Data Collection
  - Participant Observation
    - "Shadowing" developers with different roles.
  - Interview Techniques
    - 4 interviews until now ranging from 45 to 120 minutes.
- Data Collected
  - Several artifacts collected
  - What developers do, how, when, where they do, and most importantly WHY they do it.

Short Summary of Results

- Coordination using CM and bug tracking
- Use of Conventions
- Communication using E-mail
  - Problematic in some cases; but
  - Provides awareness of others work.
- Intense Parallel Development

Initial Results

- However, the conventions adopted by the developers are not automated.
- Examples:
  - Previous naming convention;
  - E-mail sent by developers right before the check-in.

Future Work

- Data Collection for 3 more weeks.
- Analysis of the data
  - Grounded Theory
  - Brahms multi-agent model
- Ultimate goal:
  - Identify requirements for technology support for this group.
  - If necessary, develop this technology.

Initial Results

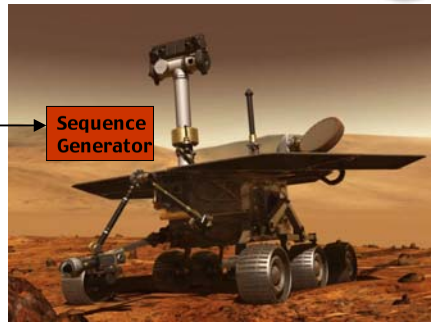
- Important communication using e-mail:
  - Is it the most effective tool to provide notifications?
  - On the other hand, e-mail is also used as a learning tool by new developers, so that they can be aware who is responsible for what process. This information is later used when one has to fix a bug in that process.

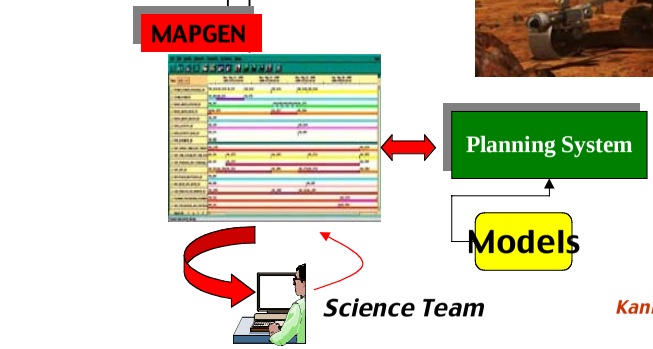
Conclusions

- CTAS:
  - Successful project developed at NASA/Ames.
- Methods
- Initial results
  - Important tools used by the developers; and
  - Problems with these tools
- Future Work

Ames Research Center

MAPGEN for MER





Alan Baba	JPL
John Bresina	ARC
Len Charest	JPL
Will Edgington	ARC
Ari Jonsson	ARC
Adans Ko	JPL
Bob Kanefsky	ARC
Pierre Maldague	JPL
Paul Morris	ARC
Kanna Rajan	ARC

Ames Research Center

How do we go where no one has gone before?

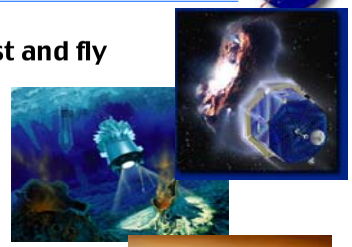
Kanna Rajan
Group Lead Spacecraft Autonomy
Autonomy and Robotics Area
NASA Ames Research Center
(Kanna.Rajan@arc.nasa.gov)

Kanna Rajan Collab/SW Eng WS 8/5/02

Ames Research Center

Current Drivers

- Shorter lead times to design, build, test and fly
- Deep(er) space missions
- More complexity
  - Visiting comets (CONTOUR)
  - Long range traverses on Mars (MER, MSL)
  - Distributed Spacecraft (LISA, DS3, TPF)
- More science return
- Tighter budgets
- New hardware certification takes time



- Software Reuse
- Newer (better?) software methodologies
- Complexity of software has risen
- Verification and Validation is critical
- Inter-center collaboration

Ames Research Center

Douglas E. Bernard	JPL
Steve A. Chien	JPL
Scott Davies	CMU
Gregory A. Dorais	Ames
Richard Doyle	JPL
Dan Dvorak	JPL
Charles Fry	Ames
Edward B. Gamble Jr.	JPL
Erann Gat	JPL
Bob Kanefsky	Ames
Ron Keesing	Ames
Jim Kurien	Ames
Guy K. Man	JPL
William Millar	**Ames**
Sunil Mohan	Ames
Paul Morris	Ames
Nicola Muscettola	Ames
P. Pandurang Nayak	Ames
Barney Pell	Ames
Christian Plaunt	Ames
Greg Rabideau	JPL
Kanna Rajan	Ames
Nicholas Rouquette	JPL
Scott Sawyer	Ames
Reid Simmons	CMU
Benjamin Smith	JPL
Gregg Swietek	Ames
William Taylor	Ames
Yu-Wen Tung	JPL
Michael Wagner	Ames
Greg Whelan	CMU
Brian C. Williams	Ames
David Yan	JPL

REMOTE AGENT EXPERIMENT

http://ic.arc.nasa.gov/projects/remote-agent/

- Remote Agent Experiment May 17-21, 1999
- Remote Agent on DS1 wins NASA's 1999 *Software of the Year Award.*

Kanna Rajan Collab/SW Eng WS 8/5/02

Ames Research Center

JPL

Results

- Testing has become central in s/w development
- Notion of a "process" to design/build/test/deploy
- Notion of model-based approaches on the increase
- Knowledge Acquisition methods are becoming crucial
- Struggle to find ways to collaborate across time-zones and cultures
  - MER
    - 10's of scientists during the design phase
    - 300+ mission staff (JPL)
    - 200+ scientists during mission ops

Ames Research Center

Needs

- Formal methods for V&V for autonomous systems
- Tools for Elicitation
- Rich array of methods for software synthesis
- Collaborative methods for design

Kanna Rajan Collab/SW Eng WS
8/5/02

Software Development
Formal Peer Inspection (cont'd)

- Participants
  - Different roles (e.g. Work Product Author, Moderator, Product Area Lead, Inspector)
  - Different responsibilities for each role
- Inspection documents and tools
  - Different formats and structures
  - Located on different platforms
  - Located in different physical and logical locations

Formal Peer Inspection Information Architecture

Gilda Pour, Ph.D., D.Eng.

Collaborative Software Engineering Tools Workshop

Formal Peer Inspection

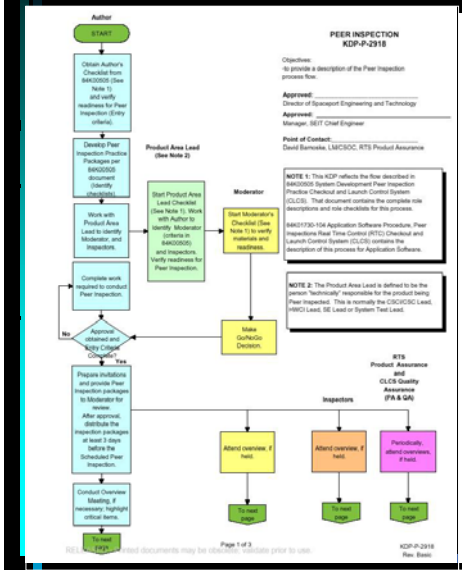
Checkout & Launch Control System (CLCS)

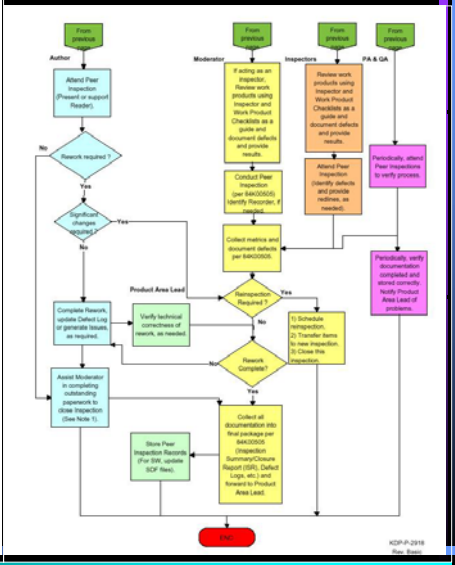
Software Development
Formal Peer Inspection

Helps mitigate risks in software development and improve software quality

Approach

- Component-based software engineering
- XML schema and XSL





Inspection Documents

- Role-based Inspection Checklists
  - Work Product Author's Inspection Checklist
  - Inspector's Inspection Checklist
  - Product Area Lead Inspection Checklist
  - Inspection Moderator's Inspection Checklist

Objective

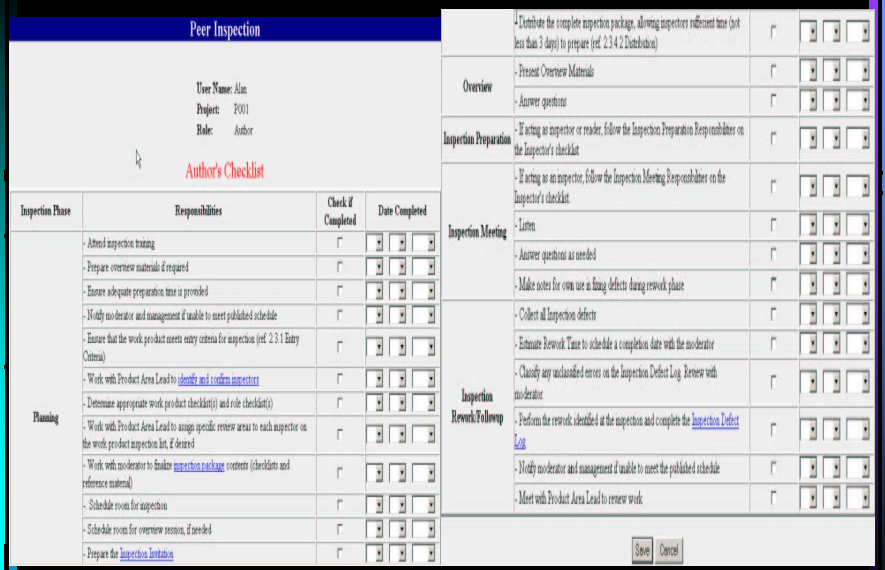
To develop a customizable, extensible, and flexible Web-based architecture to support software development formal peer inspection

Example
Role-Based Inspection Checklist Component

- Role of the inspection participant (Variables include work product author, inspector, product area lead inspector, and moderator.)
- Inspection phase (Variables include planning, overview, inspection preparation, inspection meeting, inspection rework/follow-up, and project analysis.)
- Responsibilities of a specific role in various inspection phases
- Project identification number (ID)
- Work product being inspected (Variables include documents, concept definitions, requirements specifications, top-level design, detailed design, code, test plans, test procedures, test tools, and test scripts.)
- Name of the inspection participant
- Date of completion for each task listed under "Responsibilities"
- Inspection location

Inspection Documents (cont'd)

- Work Product Inspection Checklists
  - Document-Work Product Checklist
  - Requirement's Specification-Work Product Inspection Checklist
  - High-Level Design - Work Product Inspection Checklist
  - Detailed Design - Work Product Inspection Checklist
  - Code Inspection - Work Product Inspection Checklist
  - Test Plan - Work Product Inspection Checklist
  - Test Procedure – Work Product Inspection Checklist



Peer Inspection

User Name: Alan
Project: P001
Role: Author

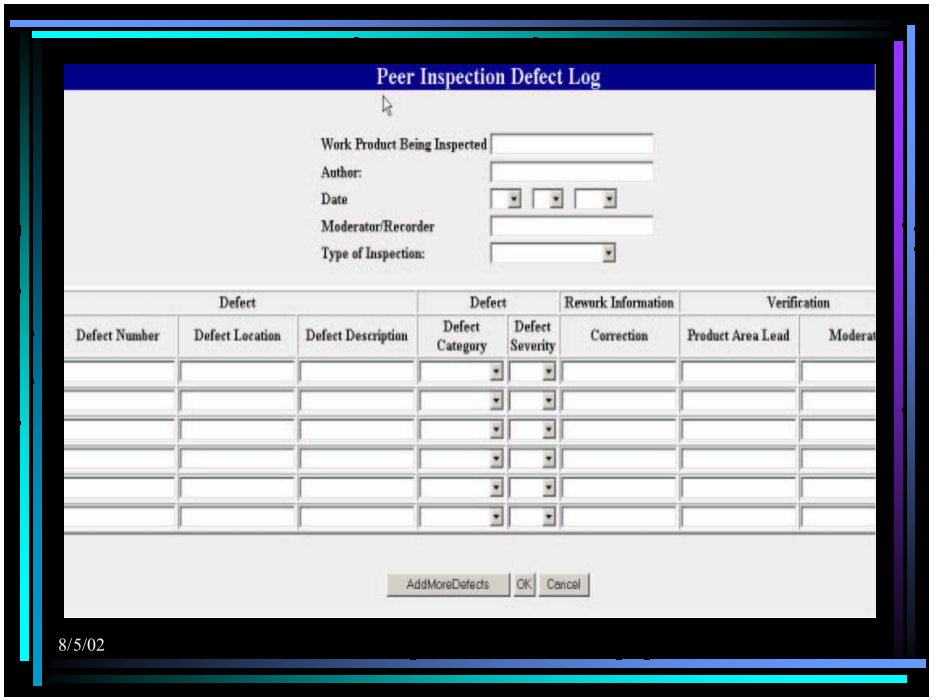
Author's Checklist

Inspection Phase	Responsibilities	Check if Completed	Date Completed
Planning	- Attend inspection training		
	- Prepare overview materials if required		
	- Ensure adequate preparation time is provided		
	- Notify moderator and management if unable to meet published schedule		
	- Ensure that the work product meets entry criteria for inspection (ref: 2.3.1 Entry Criteria)		
	- Work with Product Area Lead to identify and confirm inspectors		
	- Determine appropriate work product checklist(s) and role checklist(s)		
	- Work with Product Area Lead to assign specific review areas to each inspector on the work product inspection list, if desired		
	- Work with moderator to finalize inspection package contents (checklists and reference material)		
	- Schedule room for inspection		
	- Schedule room for overview session, if needed		
	- Prepare the Inspection Invitation		
	- Distribute the complete inspection package, allowing inspectors sufficient time (not less than 3 days) to prepare (ref: 2.3.4.2 Distribution)		
Overview	- Present Overview Materials		
	- Answer questions		
Inspection Preparation	- If acting as inspector or reader, follow the Inspection Preparation Responsibilities on the Inspector's checklist		
Inspection Meeting	- If acting as an inspector, follow the Inspection Meeting Responsibilities on the Inspector's checklist		
	- Listen		
	- Answer questions as needed		
	- Make notes for own use in fixing defects during rework phase		
Inspection Rework/Followup	- Collect all Inspection defects		
	- Estimate Rework Time to schedule a completion date with the moderator		
	- Classify any unclassified errors on the Inspection Defect Log. Review with moderator		
	- Perform the rework identified at the inspection and complete the Inspection Defect Log		
	- Notify moderator and management if unable to meet the published schedule		
	- Meet with Product Area Lead to review work		

Save Cancel

Inspection Documents (cont'd)

- Peer Inspection Invitation form
- Peer Inspection Planning Checklist
- Overview
- Listing of Reference Materials (e.g. Standards, User Guides)
- Summary of list of open issues that need to be reviewed and/or addressed
- Peer Inspection Defect Log form
- Inspection Summary/Closure Report
- Inspection Satisfaction Survey form



Next Phase

- Web-based enterprise architecture to support software development formal peer inspection at the enterprise level
- Location-transparent access of authorized users to inspection documents and tools at the enterprise level

Research Question

- **How can we model an organization's work practice in such a way that we include people's collaboration, "off-task" behaviors, multi-tasking, interrupted and resumed activities, informal interaction and geography?**
- Objective is to find representations and techniques for analyzing and designing work systems (socio-technical systems) holistically and human-centered

HCC: From Simulation to Implementation

ISR-NASA Workshop on Collaborative Software Engineering Tools
August 6, 2002

Maarten Sierhuis, Ph.D.
Senior Scientist
USRA/RIACS
NASA Ames Research Center
Moffett Field, CA
msierhuis@mail.arc.nasa.gov

Work Practice

Definition

*The performance of collective situated **activities** of a group of people, collaborating, communicating, and gaining experience while performing these activities synchronously or asynchronously.*

The Rocket Science of HCC

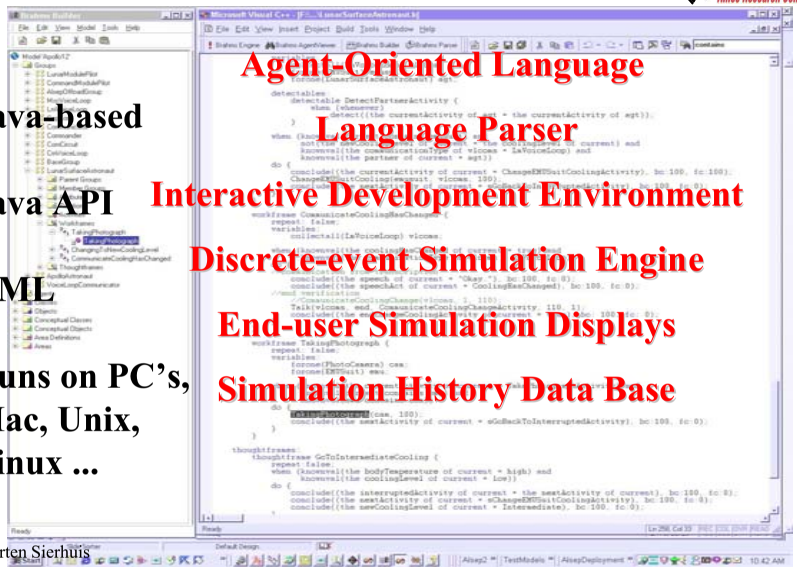
HUMAN — **COMPUTER**

What's the Delta?

Coordinate multimodal concepts, values, practice

Measure, classify, infer, depict, process

Brahms



Java-based

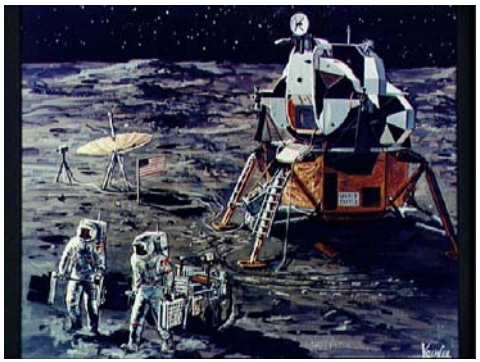
Java API

XML

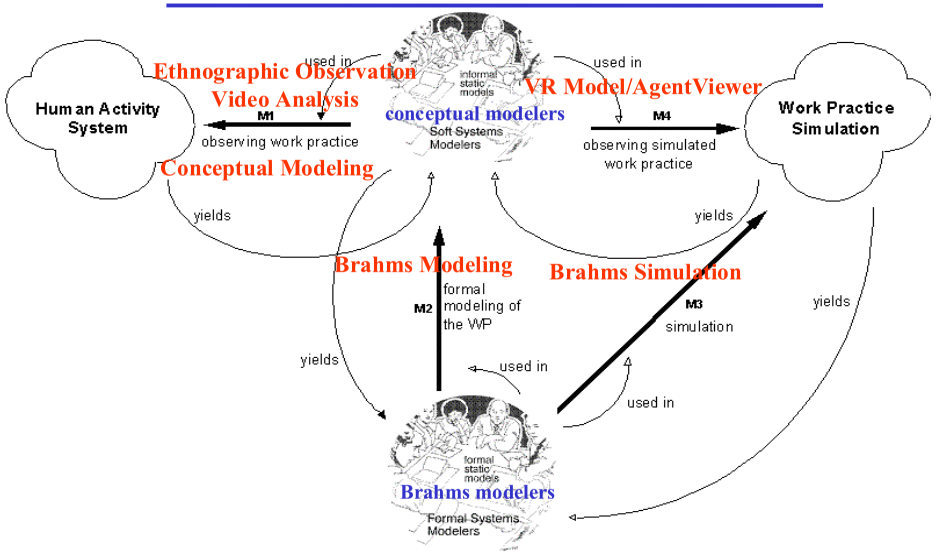
Runs on PC's, Mac, Unix, Linux ...

Work Practice Modeling

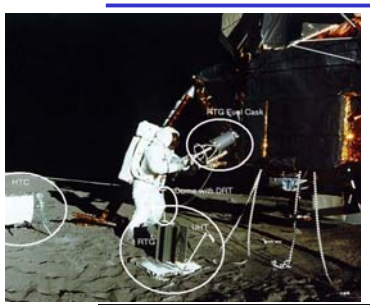
- **Groups & Agents**
  - work as activities
  - beliefs trigger work
  - bounded rationality is socially and culturally defined
- **Collaboration between Agents**
  - agents react to and interact with other agents
  - same time/same place
  - same time/different place
  - different time/same place
  - different time/different place



Collaborative Modeling



WPM cont'd

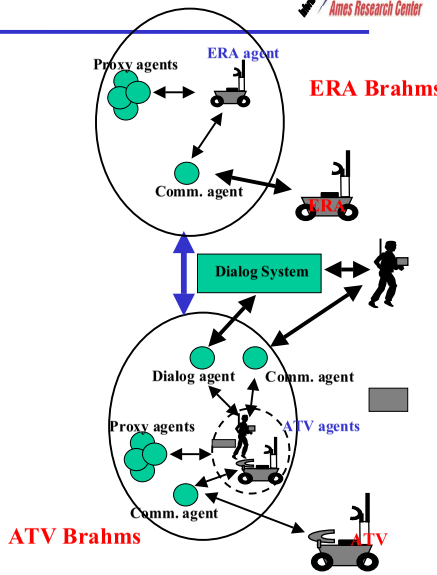


- **Tools & Artifacts**
  - tools used in activities
  - artifacts created in activities
- **Environment/Geography**
  - agents have a location
  - artifacts have a location
  - detecting real-world facts
- **Communication**
  - is situated
  - the means of communication depends on the situation (e.g. voice loop, f2f communication, telephone, faxing, e-mail)
  - impacts efficiency of work

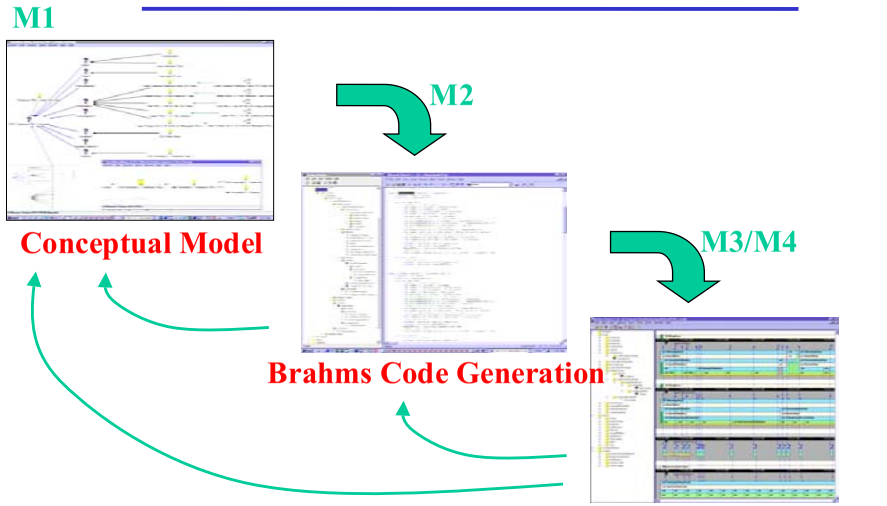
Mobile Agents

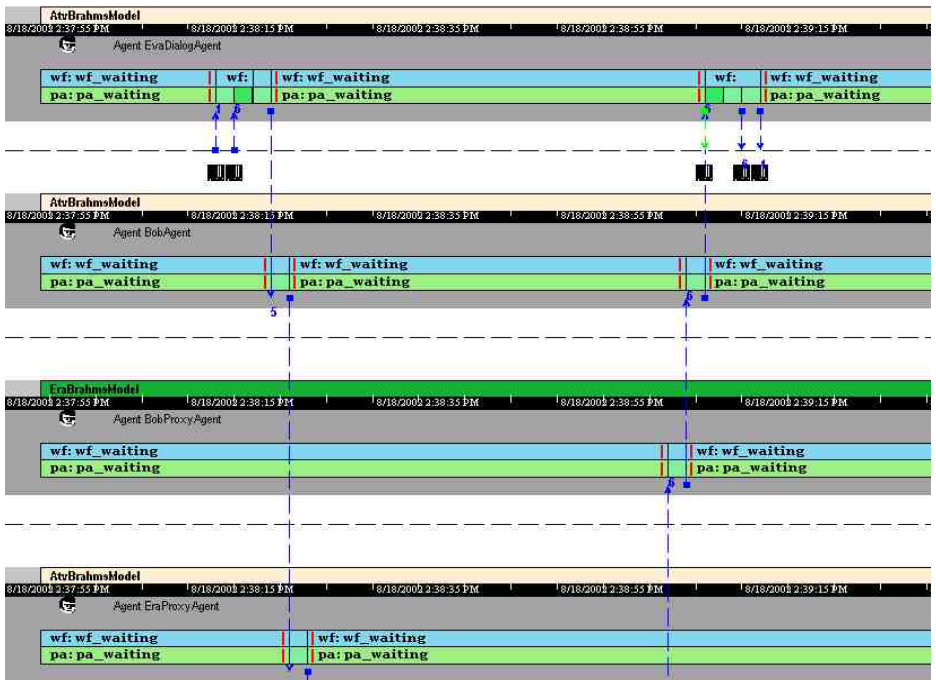
Runtime
- Models of People & Robots
- Intelligent Assistants
  - Agents for people & robots
- Software Agents
  - Dialog agent
  - Communication Agent
  - Proxy agent

Design Phase
- Simulated People & Robots
- Intelligent Assistants
- Software Agents (simulated)
- Software Systems (simulated)



Design, Generate and Simulate

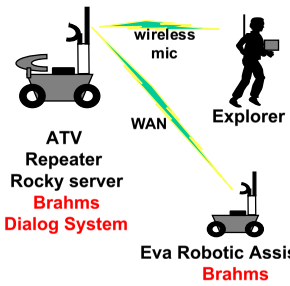




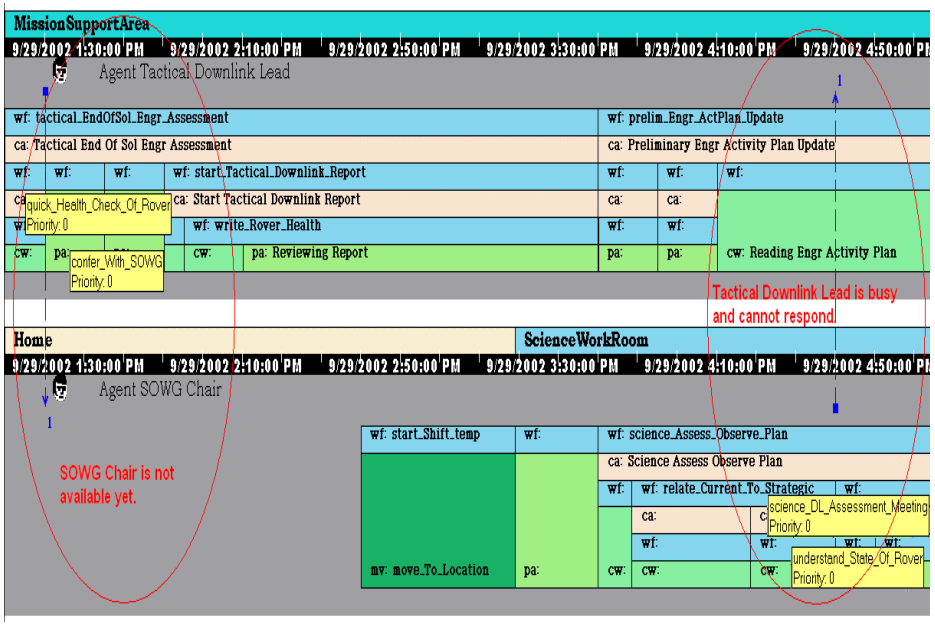
Brahms Research Projects at NASA

- Human-Robotic Teamwork
  - Teamwork & Work Practice onboard the ISS
  - Mobile Agents supporting of Mars Exploration
- Mars Habitat
  - Living and working on Mars
- '03 Mars Exploration Rover
  - Use Brahms to model Mission Operations Work System at JPL

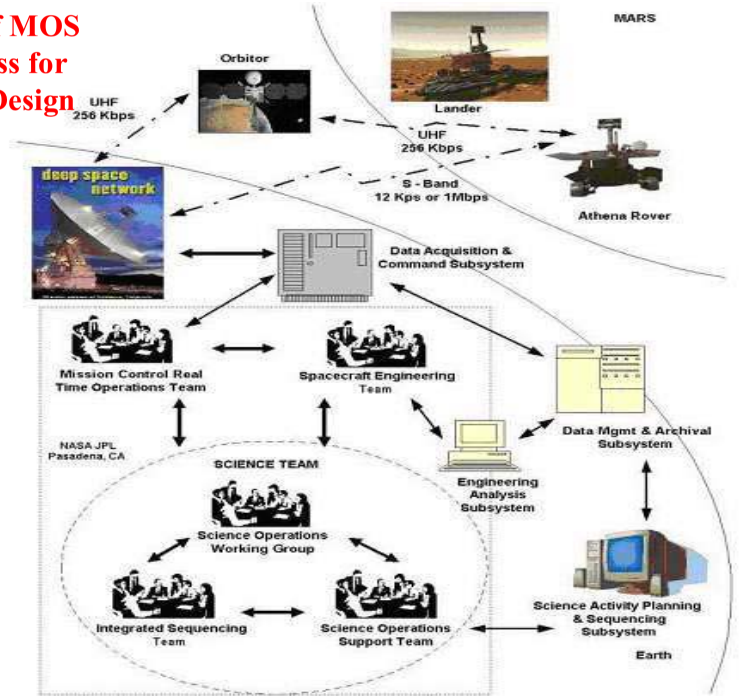
Collaborative Design with MER MOS DT



Mobile Agents Architecture

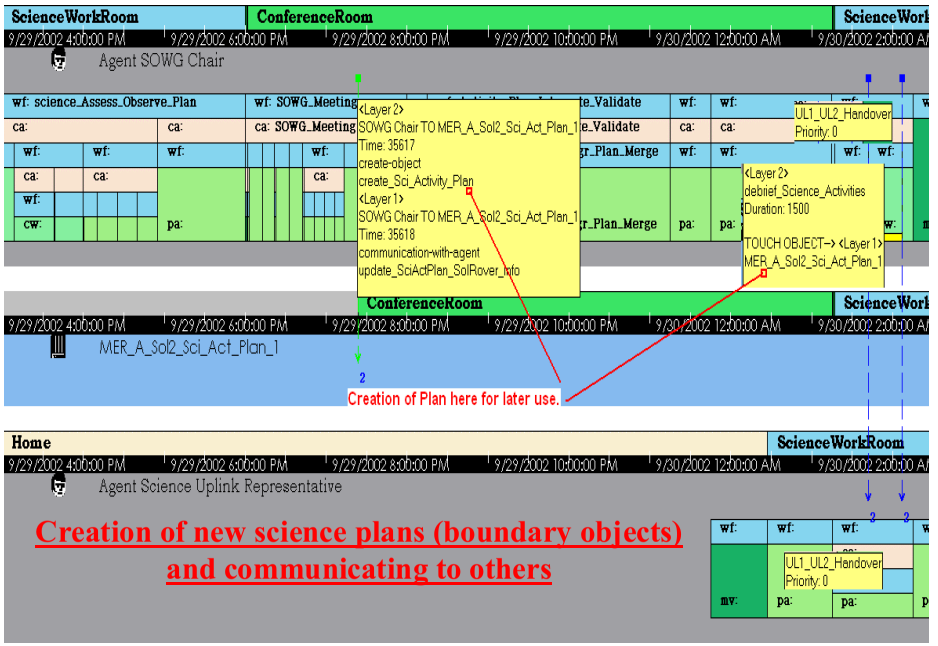


Simulation of MOS Work Process for MER MOS Design



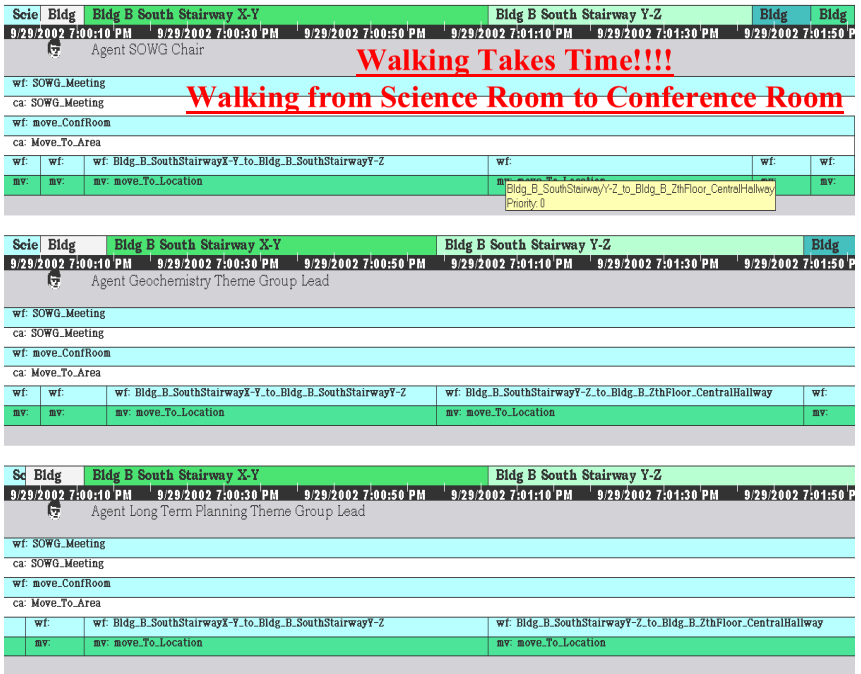
ISS Project Goals

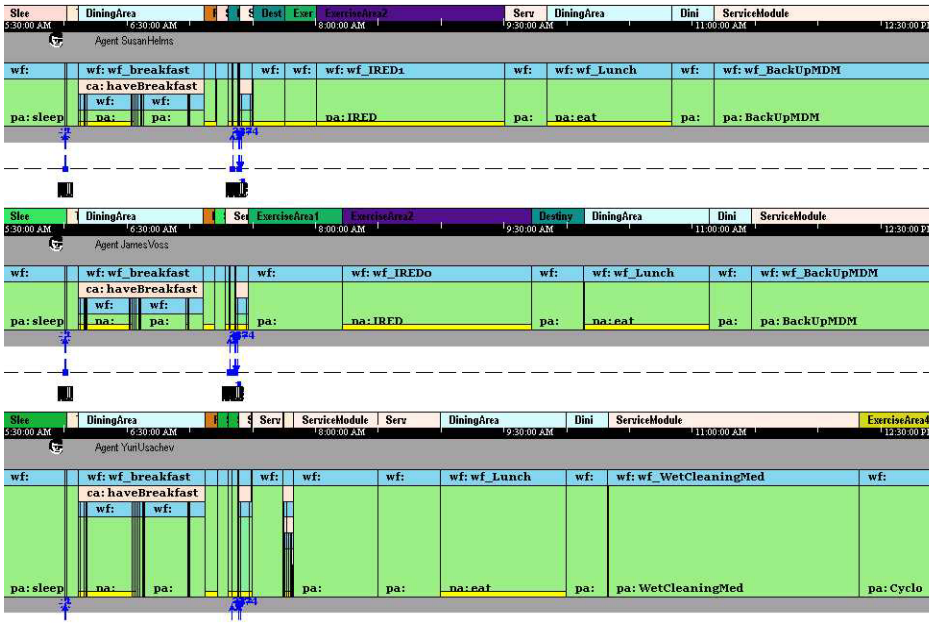
- *External goals*
  - *artifact* to study and understand work practices and teamwork onboard ISS
  - *model* to be used in:
    - *planning.* ISS Mission planning and procedure development
    - *execution.* HCI: Autonomous Intelligent Software or Robotic Agents (e.g. PSA, Robonaut) in support of teamwork .
- *Internal goals*
  - Explore use of Brahms in representing manned space missions
  - Study Brahms as an *ABSS* (Davidsson, 2002)



Modeling the ISS crew with Brahms

- Data
  - Videos, pictures, interviews
  - JSC manuals, procedures
  - Timelines, schedules
  - On-orbit and post-orbit debriefs
  - MOD servers
- Work practice data analysis
- Conceptual modeling
- Brahms model and simulation





Morning Activities ISS Exp 2, May 7, 2001

Some of the Challenges

- High communication overhead
- Shifting/rotating schedules
- Data navigation and assimilation
- Maintaining situational awareness
- Time sensitive nature of mission operations
- Heterogeneous computing environment
- Security!

Event Notification and Messaging Architectures for Real-Time Science Coordination

Elias Sinderson
elias@cse.ucsc.edu
elias@email.arc.nasa.gov
UC Santa Cruz / NASA Ames

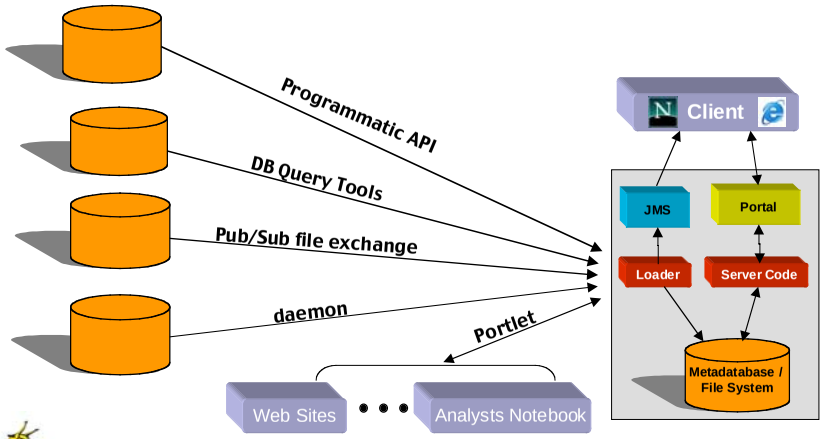
Requirements and Proposed Solutions

- Provide 'one stop' access to multiple repositories and data analysis tools under a common, Web-based interface
- Notification of 'active' resources...
- Increase overall awareness of mission personnel:
  - Scheduling tools
  - Data navigation tools
  - Mission scorecards
  - News broadcasts

Terms and Definitions

- Collaboration vs. Coordination
  - Related, but useful to distinguish between the two
  - Collaboration is when people work together on a given task
  - Coordination implies that multiple, interdependent tasks exist
- Real-time science
  - Hard deadlines
  - Closed control or feedback loop
  - Example: Remote operation of a science platform such as a satellite, space probe or robot

A Simple (?) Example

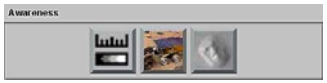


Event Notification / Messaging

- Tradeoffs between expressiveness and scalability need to be reconciled
- Heterogeneous nature of data repositories and legacy systems makes instrumenting them difficult
- Need for a complete and robust domain model

Change Awareness Dashboard

- Active resources are the primary objects of interest
- Easy access to resources
- Minimally invasive
- Peripheral awareness
- User preferences
- Subscription persistence
  - Reestablishes sessions when users start their shift
- Notification persistence
  - Keeps users up to date with any changes since they last logged on



Event Notification / Messaging

- Remote file systems can be monitored and logged with utilities such as nfslogd, auditd, etc.
- Some databases support stored procedures
- Push rather than pull information wherever possible to minimize load on system and network

Future Activities

- Finish the implementation over the next year
- Collect user feedback on system
  - Validation of GUI design
  - Comparisons with other missions
- Workflow analysis of Ops environment
- Develop web services for mobile and handheld devices
- Extend system to support multiple sites

Example: Group Awareness through a Portholes System[GLT99]

- Shows presence of collaborators and relevant spaces
- Uses visual cues (such as this theater style) to condense view according to relevance.

Using Event Notification Servers to Support Awareness

David Redmiles
Associate Professor

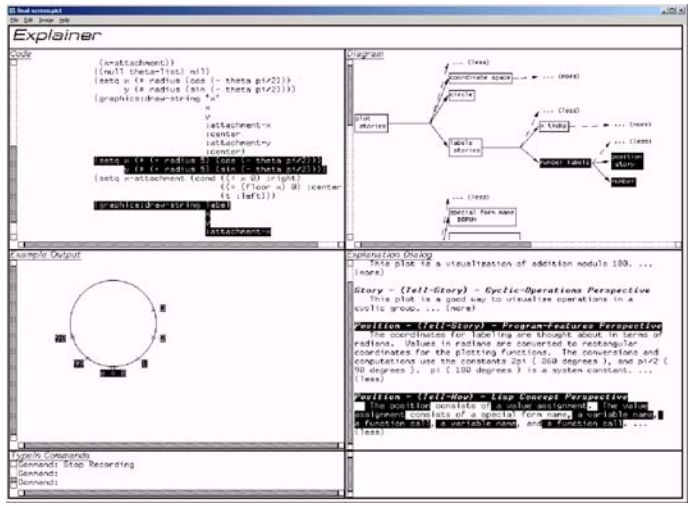
Cleidson R. B. De Souza, Santhoshi D. B., Roberto S. S. Filho
Graduate Student Researchers

Max Slabyak
Undergraduate Student Researcher

Michael Kantor (PhD '01)
Post Doctoral Student

Explainer [Red93]

- Early Hypermedia for human learning by example
- Extreme hyper-granularity
- Incremental [Minimal] Explanation
- Human Variability Reduced



Awareness and Collaboration

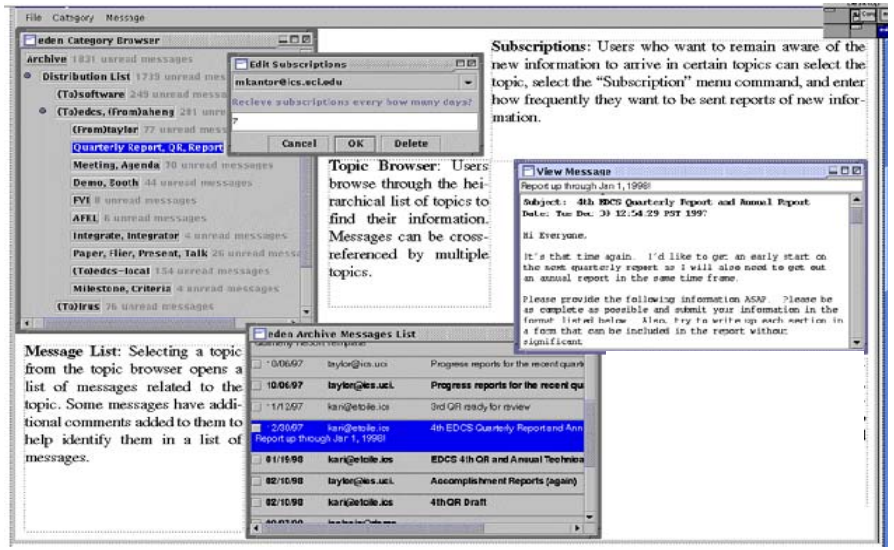
- In general, awareness means having information about other activities that affects a person's own work [DB92].
- Some types of awareness
  - Group awareness
    - Who is around and what roughly are they doing?
    - e.g., images relayed in Portholes
  - Project awareness
    - What knowledge affects (e.g., decisions are made about) project content?
    - e.g., subscriptions in Knowledge Depot
  - Application awareness
    - What's going on in running software?
    - E.g., architectural gauge

Knowledge Depot [KRZ97]

Looking for job!!!

Kantor, Redmiles, Zimmermann 97

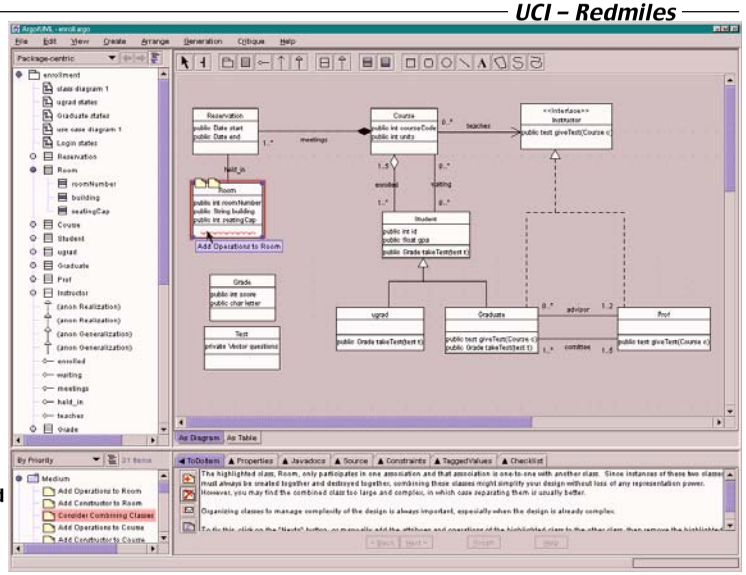
UCI – Redmiles



Argo/UML [RHR98] [RR00]

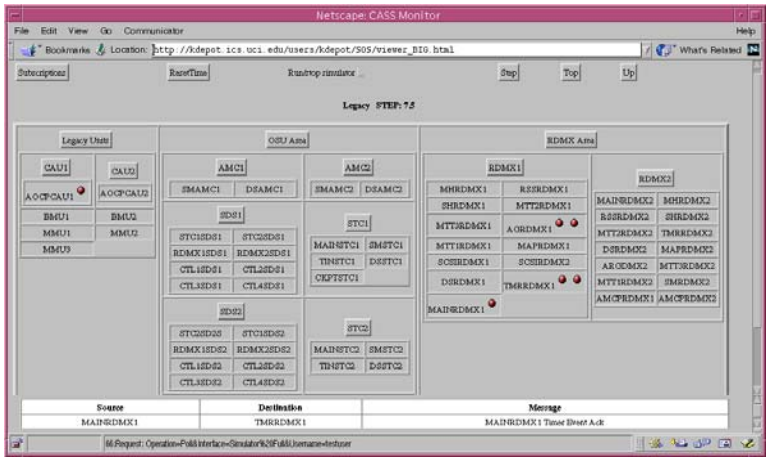
UCI – Redmiles

While designers work, design critics analyze the design and provide helpful advice. The "to do" list (lower left) presents and organizes advice about pending design changes.



Gauges for Application Awareness [SBR02]

UCI – Redmiles



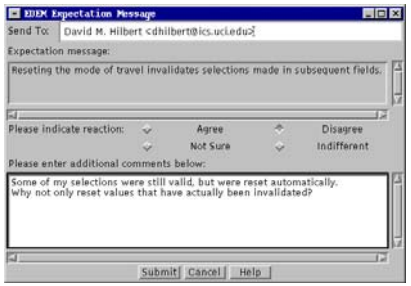
Architectural Message Passing Monitor

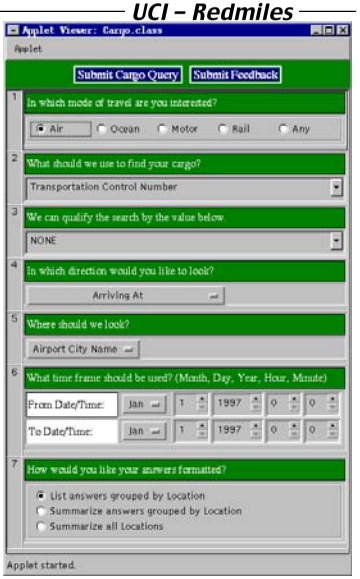
EDEM – Expectation-Driven Event Monitoring [HR98]

UCI – Redmiles

Agents monitor application events

And optionally facilitate feedback.

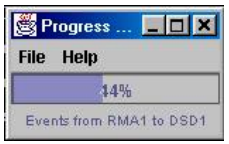




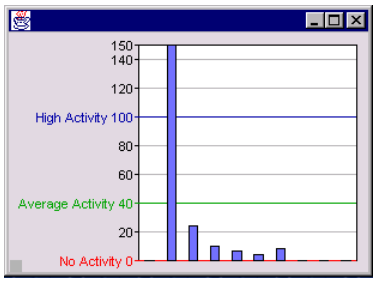
Theme – Awareness Information

- Argo/UML
  - Critics notify end users of design problems
- EDEM
  - Agents monitor application usage and report data to designers
- Knowledge Depot
  - End users subscribe to email categories / topics
- Gauges
  - "Probes" (instrumentation) should collect specific information about distributed applications' behavior and performance and supply this information to narrow-purposed "Gauges" (visualizations)
- Activity Theory
  - Many people and things affect the achievement of an objective.

Some More Gauges [SBR02]

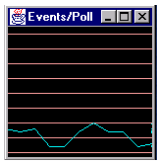


a) Progress Bar



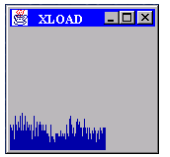
b) Bar Chart

f) Signal



c) Line Graph

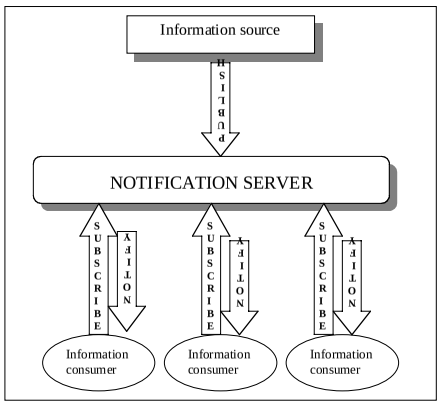
d) Clock



e) Load

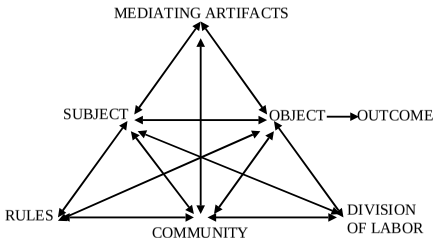
Event Notification Service

- Information Sources
- Information Consumers (Gauges)
- Event Notification Servers
- Event Services
  - Publish (Post)
  - Notify (Receive)
  - Subscribe



Activity Theory and Design [Red02a] [CSR02]

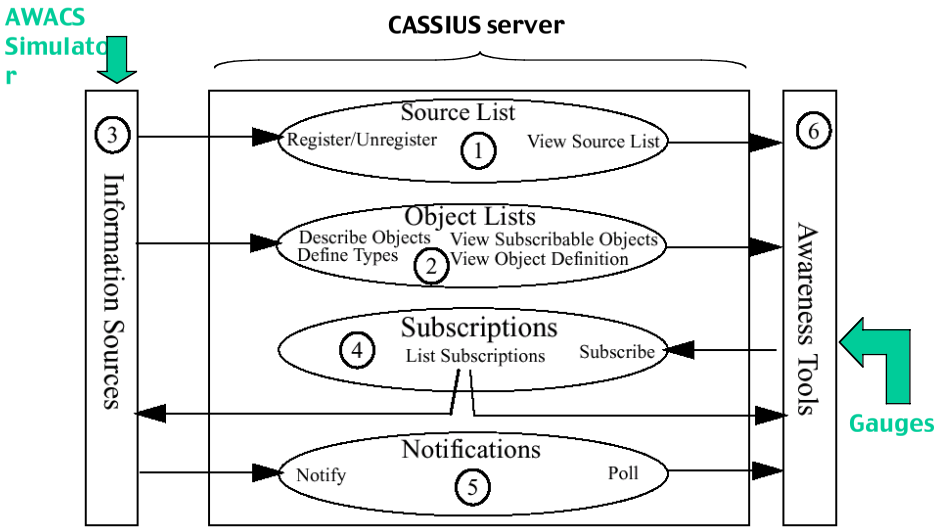
- Identify the stakeholders in the process.
- Help ensure that technology is designed to the users, other stakeholders, and the organization.
- Work toward alignment between users' rewards and business' needs.
- Work toward alignment between the rewards of the designers of the device and both the end users' and business' needs.



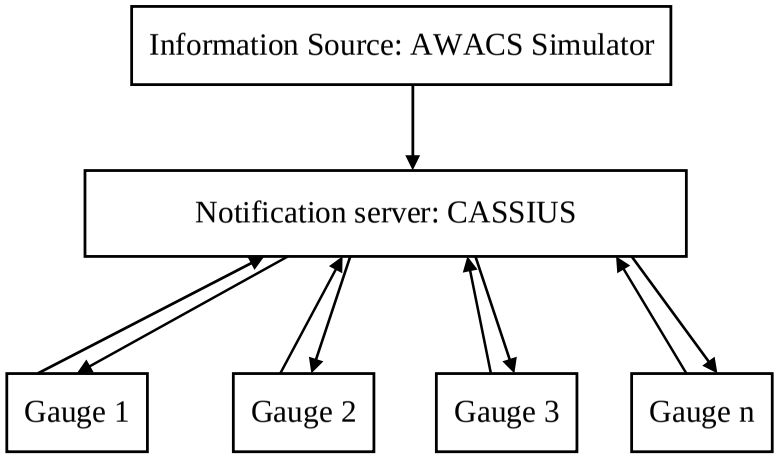
Engeström [Eng90] Activity System Model

CASSIUS [Kaiser]

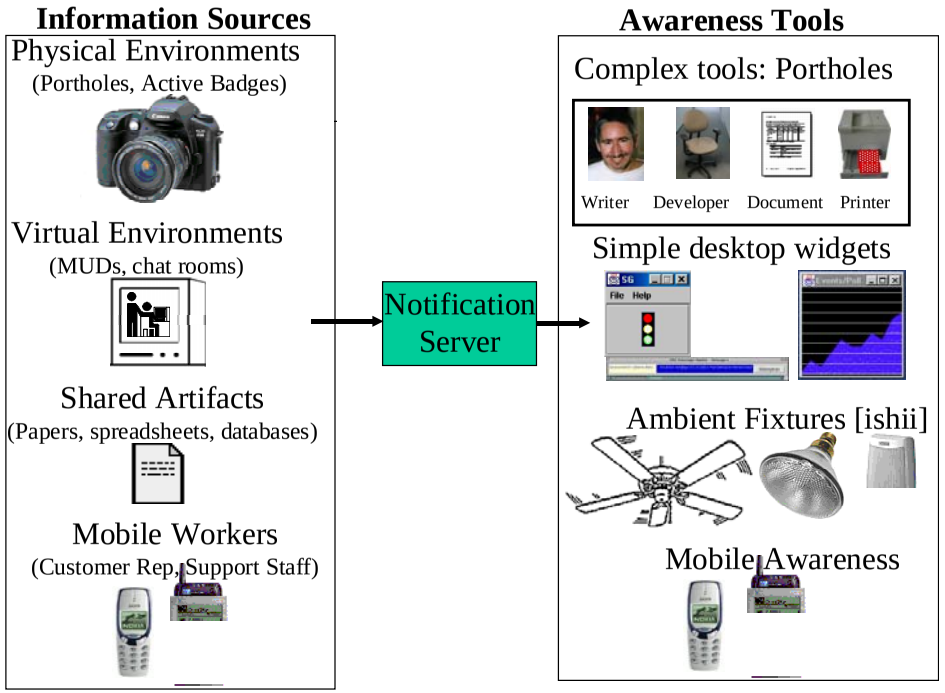
Interchangeability and the Detail-Variety Tradeoff



Using the CASS Strategy



How easy is it to provide integrated awareness?



In other words ...

- A greater variety of awareness *devices ...*
  - Critics
  - Usability expectations
  - Email notifications
  - Application gauges
  - *Security and privacy gauges?*
  - *Portholes?*

... integrated through an event notification infrastructure

Issues

- How gauges are notified about the events or The Issue of Push vs. Pull Architectures between the notification server and the gauges?
- How powerful is the subscription service for each notification server? What are the types of matching supported?
- Which objects can send events to the notification server, or Issues about event and object registration?
- Which meta-information is associated to the events sent to the notification server or How powerful are the events?
- What are the interfaces implemented by the notification servers, or how easy to change from a notification server to another?

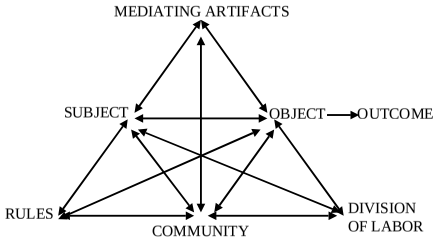
References

Conclusions

- The available software (e.g., notification servers) for building systems incorporating awareness information is very low-level and prone to design and programming errors.
- Support for complex, heterogeneous systems (e.g., multiple different, servers, information sources, and consumers) varies, currently designers must expend extra effort to design for change and flexibility.
- The goal is usability—we seek to provide a set of usable and useful services and strategy to provide usable and useful awareness capabilities for applications.

Ethnography + SE: Activity Theory

- Subjects are people within a community that work with objects to obtain an outcome.
- Rules determine the behavior of subjects and their interaction with objects.
- Division of labor determines who performs what actions.
- Mediating artifacts help subjects manipulate objects and obtain outcomes.
- Mediating artifacts have a history with respect to a community.



Engeström Activity System Model

References

- Girgensohn, A., Lee, A., Turner, T. Being in Public and Reciprocity: Design for Portholes and User Preference, In Human-Computer Interaction INTERACT '99, IOS Press, pp. 458-465, 1999.
- Redmiles, D. Reducing the Variability of Programmers' Performance Through Explained Examples, Human Factors in Computing Systems, INTERCHI '93 Conference Proceedings (Amsterdam, The Netherlands), ACM, April 1993, pp. 67-73.
- Robbins, J., Hilbert, D., Redmiles, D. Extending Design Environments to Software Architecture Design, Automated Software Engineering, Vol. 5, No. 3, July 1998, pp. 261-290
- Robbins, J., Redmiles, D. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML, Information and Software Technology, Vol. 42, No.2, January 2000, pp.79-89.
- Hilbert, D., Redmiles, D. An Approach to Large-Scale Collection of Application Usage Data Over the Internet, Proceedings of the Twentieth International Conference on Software Engineering (ICSE '98, Kyoto, Japan), IEEE Computer Society Press, April 19-25, 1998, pp. 136-145.
- Kantor, M., Zimmermann, B., Redmiles, D. From Group Memory to Project Awareness Through Use of the Knowledge Depot, Proceedings of the 1997 California Software Symposium (Irvine, CA), UCI Irvine Research Unit in Software, Irvine, CA, November 7, 1997, pp. 19-26.
- Kantor, M., Redmiles, D. Creating an Infrastructure for Ubiquitous Awareness, Eight IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001, Tokyo, Japan), July 2001, pp. 431-438.
- de Souza,C.R.B., Basaveswara, S.D., Redmiles, D. Using Event Notification Servers to Support Application Awareness, to appear in the Proceedings of the IASTED International Conference on Software Engineering and Applications (Cambridge, MA), November 2002, to appear.
- Collins, P., Shukla, S., Redmiles, D. Activity Theory and System Design: A View from the Trenches, Computer-supported Cooperative Work, Special Issue on Activity Theory and the Practice of Design, 2002, pp. ??-??.
- Y. Engeström. When is a Tool? Multiple Meanings of Artifacts in Human Activity, Chapter 8 of Learning, Working and Imagining, Painettu Kirjapaino Oma Ky:ssä, Jyväskylässä, 1990, pp. 171-195.
- Redmiles, D. Introduction to the Special Issue of CSCW on Activity Theory and the Practice of Design, Computer-supported Cooperative Work, Special Issue on Activity Theory and the Practice of Design, accepted and to appear, 2002.
- Redmiles, D. Supporting the End Users' Views, The Advanced Visual Interfaces International Working Conference (AVI 2002, Trento, Italy), May 22-24, 2002, in press.

Extra Slides

Part 2: Follow-up Materials

Citations for Follow-Up Materials

de Souza, C.R.B., Penix, J., Sierhuis, M., Redmiles, D. Analysis of Work Practices of a Collaborative Software Development Team, International Symposium on Empirical software Engineering (ISESP 2002, Nara, Japan), V. II, October 2002.

Kantor, M., Redmiles, D. CASSIUS: Designing Dynamic Subscription and Awareness Services, Workshop on Ad hoc Communications and Collaboration in Ubiquitous Computing Environments, ACM Conference on Computer Supported Cooperative Work (CSCW 2002— New Orleans, LA), November 2002, available at http://www.cs.uoregon.edu/research/wearables/cscw2002ws/papers/Kantor.pdf.

Silva Filho, R.S., Slabyak, M., Redmiles, D.F. A Web-based Infrastructure for Group Awareness based on Events, Workshop on Network Services for Groupware, ACM Conference on Computer Supported Cooperative Work (CSCW 2002—New Orleans, LA), November 2002, available at http://awareness.ics.uci.edu/~rsilvafi/papers/Workshops/CSCW2002-workshop.pdf.

de Souza, C.R.B., Redmiles, D.F., Mark, G., Penix, J., Sierhuis, M. Management of Interdependencies in Collaborative Software Development, ACM-IEEE International Symposium on Empirical Software Engineering (ISESE 2003), September 2003.

de Souza, C.R.B., Redmiles, D.F., Opportunities for Extending Activity Theory for Studying Collaborative Software Development, Workshop on Applying Activity Theory to CSCW Research and Practice, in conjunction with the 8th European Conference of Computer-Supported Cooperative Work (ECSCW 2003—Helsinki, Finland), September 2003, available at http://www.uku.fi/atkk/actad/ecscw2003-at/desouza+redmiles.pdf.

de Souza, C.R.B., Redmiles, D.F., 'Breaking the Code', Private and Public Work in Collaborative Software Development, Supplement Proceedings of the 8th European Conference of Computer-Supported Cooperative Work (ECSCW 2003—Helsinki, Finland), September 2003.

de Souza, C.R.B., Redmiles, D., Dourish, P., 'Breaking the Code', Private and Public Work in Collaborative Software Development, International Conference on Supporting Group Work (Group 2003—Sanibel Island, FL), November 2003.

Analysis of Work Practices of a Collaborative Software Development Team

Cleidson R. B. de Souza[1,3] John Penix[2] Marteen Sierhuis[3] David Redmiles[1]

[1]*Department of Information and Computer Science*
University of California, Irvine
Irvine, CA, USA

[2]*Computacional Sciences Division*
NASA Ames Research Center
Moffett Field, CA, USA

[3]*Research Institute for Advanced Computer Science*
NASA Ames Research Center
Moffett Field, CA, USA

Abstract

This paper reports preliminary results of a field study of a software development team. This team develops a suite of tools called CTAS, designed to help air traffic controllers manage complex air traffic flows at large airports. We observed that CTAS developers employ two main tools for coordinating their work: a configuration management system and a bug tracking system. These tools allow them to coordinate their activities supporting a high level of parallel development. Communication and cooperation among developers with different roles is achieved using product requests. Future results from our study will provide insights into the complexities of cooperative software development and help to design tools to support it.

Keywords*: Field Study, Empirical Studies, CSCW, Cooperative Work, Cooperative Software Development.*

1. Introduction

Software development is a typical cooperative activity where experts from different domains are necessary. Indeed, developers of large systems spend about 70% of their time working with others[10]. At NASA, this problem is much more difficult because of the increasing complexity of the software being developed. In fact, several efforts to improve the software engineering practices through the dissemination of best practices and infusion of new technologies are taking place at NASA. However, these efforts will only be effective if they address the real ways that people work together to develop software.

Gerson and Star[1] observe that no matter how formal and well-defined a process may seem, there is always a set of informal practices by which individuals and groups monitor and maintain the process, keep it on track, recognize opportunities for action and the necessity for intervention or deviation. In order to understand these practices, the first author conducted an eight-week qualitative study of a software development team using non-participant observation and informal interviews for data collection. This paper reports our initial findings.

2. The Setting

The group observed develops an application called CTAS (Center TRACON Automation System). CTAS is a suite of advisory tools designed to help air traffic controllers manage the complex air traffic flow at large airports. The source code is developed in C and C++ and is about 1,000 K lines long. The development team is divided in two groups: developers and the verification and validation (V&V) staff. Developers are responsible for writing new code, for performing bug fixing and enhancements, and so on. There are 25 developers, including researchers that write their own code. The V&V staff is responsible for testing the software and reporting, keeping a running version for demonstration purposes and maintaining user manuals. This group is composed of six engineers.

3. The Methods

The first author spent eight weeks at the field site. We adopted non-participant observation[3] and informal interviews[5]. In addition to field notes generated by the observations and interviews, we collected software development tool manuals, ISO 9000 procedures, product requests for software changes (PR's), and e-mails exchanged regarding the data and documents.

Initial data collection was centered on understanding the daily work of the developers. During this stage, it became clear that the configuration management (CM) and the bug tracking tools combined with e-mail communication were central to the **coordination** of activities. These results are not surprising based on previous studies by Grinter[2]. Therefore, later data collection was focused on understanding exactly how developers use these tools to perform their work. The interviews focused on observing and understanding the usage patterns of these tools.

4. Initial Findings

An important aspect in fieldwork is the interplay between data collection and analysis: data collection is directed by on-going analysis of the data[9]. Our initial re-

† The authors would like to thank the CTAS group for their help during the fieldwork and the NASA Research Grant NAG2-1555 for the financial support. The first author would also like to thank CAPES (grant BEX 1312/99-5) for the financial support. He is also a faculty at the Department of Informatics, Federal University of Pará, Belém, Pará, Brazil.

sults of this analysis are described in this section. Further analysis is still necessary to obtain more reliable results.

4.1 Parallel Development

We noted that software developers often engage in parallel development. This confirms the results from Perry *et al*.[6], but contrasts with the groups studied by Grinter[2], where developers avoided this situation. Parallel development usually happens when more than one developer has to make changes in the same file. Conflicts might occur when one of these developers check the file back in the repository, because the current version of other developer will be outdated and his modifications might be based on the code that was modified. To update his version, one only needs to merge the other changes back in his code. According to the developers, these conflicts are infrequent and not likely to occur. We plan to use log of the tool usage to test this assumption.

In order to avoid these conflicts, the group adopted the convention that before checking one file in, a developer must send an e-mail to their mailing list describing the files that were changed and the product request associated with the changes. Developers even go to their co-worker's office to talk about the changes that they made or browse the CM repository in order to understand these changes.

Another strategy used by the developers is the *partial check-in*, i.e., to check files in, even when their work is not completely finished. This strategy is employed by those who work with files that are constantly changed by several developers, which makes conflicts more likely. This helps them to prevent those conflicts and avoid several back merges to update their code.

4.2 Conventions

The team studied adopts several conventions in order to cooperate effectively. Conventions are rules or arrangements established in the group, common and accessible to its members[4]. Examples of such conventions are the e-mail that has to be sent before the check-in, or the naming conventions that must be followed when dealing with the CM and bug tracking tools. However, these conventions are not properly supported by their tools which is a source of complaints by the developers. For example, the creation of branches in the CM tool must be based on the PR number recorded in the bug tracking tool. This creation is a cumbersome process that could be easily automated since this is a standard procedure.

4.3 Product Requests (PR's) as Boundary Objects

During the fieldwork, we also identified that PR's are used as boundary objects by members of the team with different roles. Boundary objects are objects both plastic enough to adapt to local needs and the constraints of the several parties employing them, yet robust enough to maintain a common identity across sites[8]. In this group, PR's are used by end-users *liaisons*, developers and testers serving for different functions. For example, when a bug is identified, it is associated with a specific PR. Whoever identified the problem is also responsible for describing '*how to repeat*' it. The developer assigned to repair the bug uses this description to identify and fix it. After that, he must fill a field in the PR that describes how the testing should be performed to properly validate the fix. This information is expanded by the test manager to create the test matrices that are later used by the testers. Another field conveys what needs to be checked by the manager when closing the PR. Therefore, it is a reminder of the aspects that need to be validated.

5. Conclusions and Future Work

Data analysis will be performed using grounded theory[9] and analytical tools like boundary objects and social networks. We also plan to use the Brahms work practice modeling and simulation method[7], in order to simulate the impact of inserting collaborative tools into the development activity. We are particularly interested in understanding the consequences of the parallel development identified in this group: reasons why these developers engage in parallel development might be social, organizational, technological or combinations there of. It is important to identify and understand these reasons so that this practice might be improved, made more effective or safely adopted by other development groups. Initial results indicate that the branching strategy employed in the CM tool properly supports such development, but this is still an open question. We collected log usage data and plan to apply statistical techniques to validate this hypothesis.

6. References

[1] Gerson, E. M. and Star, S. L. *Analyzing Due Process in the Workplace.* ACM Trans. on Office Information Systems, 1986. **4**(3): p. 257-270.

[2] Grinter, R., *Supporting Articulation Work Using Configuration Management Systems.* Journal of Computer Supported Cooperative Work, 1996. **5**(4): p. 447-465.

[3] Jorgensen, D. L., *Participant Observation: A Methodology for Human Studies*. 1989, Thousand Oaks: SAGE Publications.

[4] Mark, G., *et al. Supporting Groupware Conventions through Contextual Awareness*. in *(ECSCW'97)*. 1997, p. 253-268.

[5] McCracken, G., *The Long Interview*. 1988: SAGE Publications.

[6] Perry, D. E., Siy, H. and Votta, L. G. *Parallel Changes in Large Scale Software Development: An Observational Case Study*. in ICSE 1998.

[7] Sierhuis, M., *Modeling and Simulating Work Practices. BRAHMS: a multiagent modeling and simulation language for work system analysis and desing*. 2001: Ponsen & Looijen BV.

[8] Star, S. L. and Griesemer, J. R. *Institutional Ecology, Translations and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology.* Social Studies of Science, 1989. **19**.

[9] Strauss, A. and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Second. ed. 1998, Thousand Oaks: SAGE Publications.

[10] Vessey, I. and A. P. Sravanapudi, *CASE Tools as Collaborative Support Technologies.* CACM, 1995. **38**(1): pp. 83-95.

CASSIUS: Designing Dynamic Subscription and Awareness Services

Michael Kantor
Institute for Software Research
University of California, Irvine
Irvine, CA 92612 USA
mkantor@ics.uci.edu

David Redmiles
Institute for Software Research
University of California, Irvine
Irvine, CA 92612 USA
redmiles@ics.uci.edu

ABSTRACT
CASSIUS is an awareness server which assists users in designing subscriptions for maintaining awareness of events within work, physical and social environments. This environment is designed to work with a wide range of awareness tools using desktop computers, mobile devices and ambient fixtures[4]. This work investigates the requirements for creating ad-hoc subscriptions – a subscription that is created either by the user or a software agent, and which only exists for a brief period of time. Design guidelines are proposed that help address the problem inherent in having users invest effort in creating a subscription which may last for only the few minutes in which they are in a specific location or context.

Keywords
Awareness tools, Notification servers, agents, CASSIUS, ad-hoc networks

INTRODUCTION
The "Creating Awareness with Subscription Services" (CASS) strategy is an approach for creating a ubiquitous awareness environment [5]. The goal is to enhance people's ability to coordinate with various actors within work, physical and social environments by providing a usable and useful environment for awareness and coordination. The CASS strategy consists of a set of guidelines for the design of software based awareness environments.

This paper begins by presenting an overview of these guidelines and presents our implementation of this ubiquitous awareness. We then discuss potential extensions to the guidelines and implementation which address the issues of designing an awareness environment that is usable for the ad-hoc creation of subscriptions for monitoring contextual information.

CASS Guidelines
The CASS strategy consists of a set of guidelines for creating a usable and useful awareness environment. These guidelines can be divided into three categories: provide access to diverse information, remove guesswork from specifying the information of interest and support flexibility in the choice of awareness styles for representing awareness information.

LEAVE BLANK THE LAST 2.5 cm (1") OF THE LEFT COLUMN ON THE FIRST PAGE FOR THE COPYRIGHT NOTICE.

Provide Access to Diverse Information
Research in awareness technologies has focused upon tools designed to monitor a single source (or a narrowly defined set of sources) of awareness information. This has been the case because the projects were either experimental, investigating a style of presenting awareness information with some demonstration source of awareness information or because they were implemented within a context where there was only one information source that the designer was interested in.

In a ubiquitous awareness environment, an awareness tool has access to diverse sources of awareness information allowing each user to monitor the kinds of information that matter to them. This was done by the Elvin Tickertape [3] which could monitor discussion groups, news, email, and other notifications sent to the notification server. To support awareness and coordination in diverse environments, we can not limit ourselves to monitoring a single source of information. An awareness tool needs to be able to obtain information from multiple sources of awareness information, and integrate them together to give users a broader understanding of what is happening within their work, social or physical environments. Nor can we limit users by telling them that the only information that they can become aware of is news, photos of offices [1], or any other single source.

Remove Guesswork from Specifying Information of Interest
Having access to diverse sources of awareness information would be insufficient if the user does not know what sources of information are available. To provide a *usable* awareness environment, the user needs to be informed (preferably by the awareness tools rather than by coworkers) of what sources of awareness information are available, what each source monitors, and what kinds of changes and events can be detected. The awareness environment must provide users with meta information describing the awareness information accessible to the environment.

For example, if a source of information is a research paper, the sections and subsections could be monitored for changes, as could word or page counts. As a second example, if users monitor for traffic problems, they need to

know what freeways and roads are monitored and what kinds of traffic events are reported so that they can choose which ones to monitor.

Without this meta information, any attempt by the user to describe their interests involves a great deal of guesswork, leading at best to partial success, and more likely to frustration. Access to this type of information is a prerequisite for a usable ubiquitous awareness environment.

Support Choice of Awareness Styles

A common problem with awareness technologies is that they tend to provide a fixed awareness style with very little room for selection of alternatives. In this work, the term awareness style refers to the manner in which information is presented to users and varies along a variety of dimensions including:

1. Intrusive vs. peripheral dimension: how intrusive/disruptive is the presentation of new awareness information? If the goal is to be immediately notified of information as it occurs, an intrusive style is needed. If the goal is to maintain general awareness of ones environment, utilization of peripheral senses may be more appropriate.
2. Mobility dimension: Can the awareness tool be used as a person's work moves through different physical and social contexts? Can it use mobile devices, or does it require greater display, networking or computational resources? Does its presentation style require the kind of user attention only available within an office or control room?
3. Information Representation dimension: What kinds of information does the representation of the information focus upon?
4. Cognitive Effort dimension: How much effort is needed to interpret the representation?

To provide an awareness environment that is *useful*, people need to not only be able to choose what information to monitor, they need to be able to choose how to be made aware of the information. Ideally, they would have hundreds of different awareness tools to choose from, and could choose the one which best fits their work environment, work practices and their needs with respect to some subset of the information they intend to monitor.

Further, the user should not be limited to one awareness tool at a time, nor one awareness tool for any source of awareness information. As a user leaves an office setting for a meeting, lunch or other situations, the style of awareness that suits this new environment may change and the user needs to have the option of changing awareness tools to match the new environment. When the user is in the office, there may be many sources of information, one subset of which is monitored with an intrusive tool, and a second subset of which is monitored with a peripheral awareness tool.

CASSIUS

Our implementation of the CASS strategy is called CASSIUS (CASS Information Update Server). It is a notification server [7] which has been optimized for usability as an awareness server. Figure 1 shows a service-based architecture that CASSIUS implements, and Figure 2 shows an awareness source browser and subscription editor provided with our CASSandra toolkit.

As shown in the top two services of Figure 1, sources of awareness information must register with the server, listing the objects that they monitor and describing the types of events that can affect those objects. The awareness tool can then support users browsing through lists of sources of awareness information (shown in the top left column of Figure 2). For each information source, the user can browse through hierarchies of objects and properties monitored by that information source (top center column of Figure 2). When the user selects an object to monitor, lists of events that can affect the object are listed, allowing the user to optionally refine their subscription to just those types of events (top right column of Figure 2). A single awareness tool can monitor as many subscriptions and information sources as suits the user's needs and the tool's awareness style.

Representation of Arbitrary Information

A key issue in our design involves the representation of awareness information from any information source. If the designer of the awareness tool does not know in advance what the source of awareness information is, how can the tool represent that information? The answer is that all notifications, regardless of what software sent them, must be formatted using data fields that have a fixed interpretation shared by all CASSIUS awareness tools and information sources. The awareness tool need not understand the meaning of the data sent in a notification, but does need to understand its nature, that one field contains verbal/textual descriptions of the event, another field quantifies the extent of change, etc… Our design attempts to account for the information needs of a broad range of awareness styles by using the notification fields of Table 1.

Sample Applications

We currently have a WebDAV server (CassDAV), and an AWACS simulator which send notifications to CASSIUS,

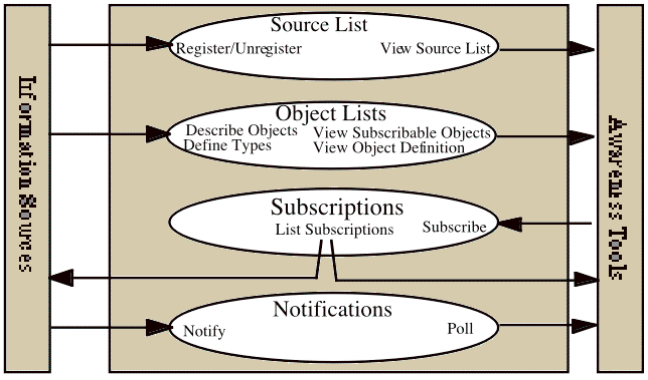


Figure 1: CASSIUS service architecture

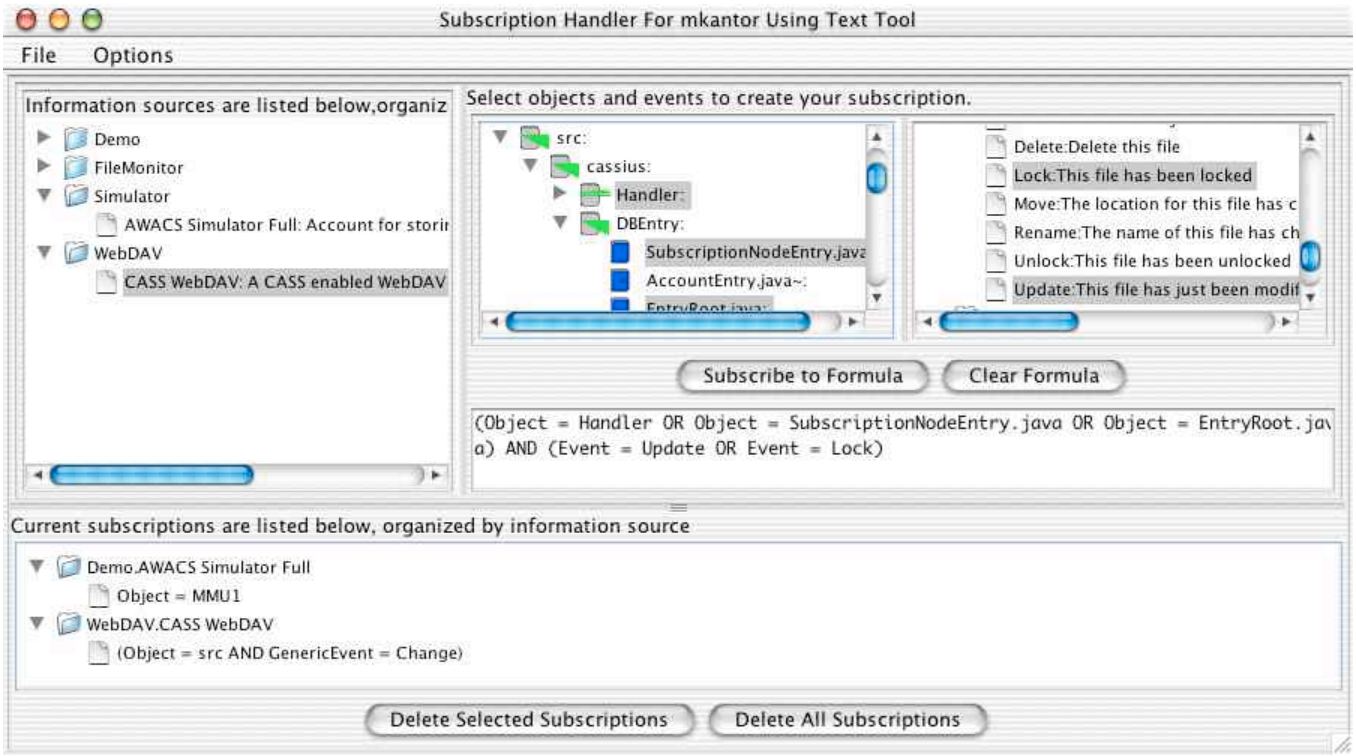


Figure 2: CASSandra information source browser and subscription editor

and we are working on a CVS repository and a Portholes implementation [1]. In the case of WebDAV and CVS repositories, the monitored objects are files and folders, which are described to the server so that the user, using an interface such as that presented in Figure 2, can browse through a representation of the file system to find and select files and folders to monitor. Notifications report on the nature and extent of the changes or operations performed upon the files and folders.

In the case of Portholes, which creates awareness by distributing photos of people at work in their offices, the monitored objects are groups and individuals, the notifications indicate the extent of changes between successive photos, and contain a URL to the photo.

To monitor these and future sources of awareness information we have a growing body of awareness tools including simpleScroller (a tickertape such as was illustrated by Elvin [3]), EventLister (a debugging tool to help developers see the notifications that their code sends) and BiffArray (Figure 3). We are also working on an email-based tool for sending digests of events, and are planning to adapt our mobile awareness technology called MiniPortholes.

BiffArray

BiffArray (Figure 3) is modeled on Xbiff, a common mail awareness indicator in unix windowing environments. It provides a row of Biffs, where the graphics within the icon show the most recent event to come from the objects being monitored. Rather than a mailbox with flag up or down graphic (as was done in XBiff), it shows the GenericEvent field (see Table 1) of the most recent notification to be received. As there are five values of Generic Event, there are 5 images used to represent the different states. Each biff in the display can be configured both in what it monitors and in what sounds it uses to notify the user [2].

Each biff can monitor a different source of information. For example, if you have six biffs, two could monitor files and folders that you work with, two could monitor coworkers, one could monitor activity on a chat group, and the last could monitor the state of your group's printer.

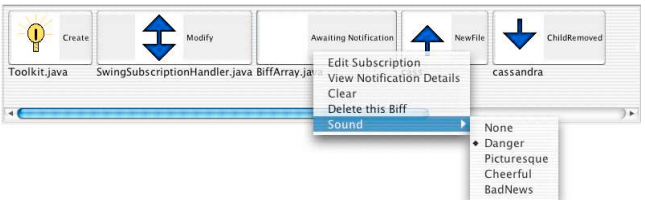


Figure 3: BiffArray: Visual and Audio Icons

Mobile Awareness

MiniPortholes (Figure 4) is a mobile awareness technology implemented in J2ME. It allows users to subscribe to maintain awareness of individuals such as coworkers and family. When this tool uses the CASSIUS server, it enables users to not only subscribe to monitor other MiniPortholes users but also monitor all types of CASSIUS information sources. This means that system administrators can monitor their servers, salesmen can monitor their inventory, parents can monitor their children, and in fact, a parent who is a system administrator and

Summary	One line textual summary of the change
GenericEvent	An event name chosen from a list of generic event names. Generic event names are shared by all information sources and enable awareness tools to understand the general nature of the event even if they can't interpret the specific nature of the event represented by the Event field. Currently supports "Activate", "Deactivate", "Increase", "Decrease" and "Change" (the last being a catch-all for events not fitting other categories).
Event	An event name specific to the information source and to a type of object within the information source. Events reporting on a section of a document might include "Text Added", "Text Removed", "Subsection Added", and "Subsection Removed".
URL	Optional link to more information about a notification. Leads users to text, images or information source specific data files.
Person	Optional person associated with event.
Place	Optional place associated with event.
Object	Identifies the object or property that has changed.
AccountPath	Identifies the information source.
NumericalValue	Optional numerical value to quantify the change.

Table 1: CASSIUS Notifications

salesman can monitor all three simultaneously – hopefully not while driving.

While currently using a simplified version of CASSIUS, we hope to integrate MiniPortholes with CASSIUS soon.

AD HOC AWARENESS INFORMATION

The high level goals of this work (the creation of a ubiquitous awareness environment) are important whether one is talking about work (awareness and coordination among coworkers, often distributed both spatially and organizationally), family (awareness and coordination with family members scattered around a city) or a physical environment (awareness of problems such as upcoming traffic, weather, riots, parades, statistics related to a sporting event you attend and special deals at your favorite coffee shop just down the street). Effective support of these diverse environments requires:

1) Creation of ad-hoc subscriptions, whose life span may be as little as 10 minutes (and where the time to specify the subscription must be comparably short).
2) Location based awareness servers that awareness tools connect to on-the-fly to discover new sources of awareness information.

The principal of what must be done remains unchanged: 1) the users must be provided with meta information telling them what information sources are available and what types of information can be subscribed to within each information source, and 2) users choose awareness styles for each type of information. However, in this new

Figure 4: MiniPortholes, mobile awareness

environment, extensions are needed in how these services are provided.

Extension 1: Detecting and Logging Information Sources

In our current implementation, users can view lists of information sources on the servers that they have permission to access. If we introduce location-based awareness servers (perhaps for sending traffic awareness to people on freeways) and time-based awareness servers (a server which only exists for a short peiod of time, such as for the duration of a county fair, or a festival), the nature of these lists must change. To effectively provide users with lists of information sources that they can monitor, mobile awareness tools need to be able to

1. Detect the presence of awareness servers as they come into range,
2. Obtain lists of information sources from these servers that users can browse through,
3. Store the lists of information sources and information about the awareness servers (such as that it was running on a traffic monitoring server, or on some stranger's PDA),
4. Categorize the stored information according to the nature of the awareness server (group all traffic awareness servers together, group all PDAs running their own servers together), and by the information source (all information sources that monitor a calendar get grouped together, regardless of what awareness server it came from).

Extension 2: Usability in Ad Hoc Subscriptions

A key issue in supporting ad hoc subscriptions is the efficiency with which the subscription can be created. How much examination of the display and selection of options must be done to allow the user to monitor traffic for the next 15 minutes? To address these problems, additional

guidelines have been created for the design of information sources and awareness tools.

Support a Spectrum of Complexity

Location and time based awareness servers should provide simple options for subscribing. While it should be possible to carefully refine long term subscriptions so that the awareness tool doesn't waste time presenting unwanted information, support is also needed for the fast and less precise task of creating short-term subscriptions.

For example, a person at a county fair can subscribe to the fair's scheduled events and be notified each time a new event is about to begin. Or the person can be more careful and look at the objects under "Scheduled Events" in the object hierarchy and subscribe to only be notified when musical events are about to start. Both subscriptions are useful. One requires more time and thought – time which people spending all day at the fair are more likely to invest than people attending for only part of the day.

This scaling is supported in CASSIUS in the form of notifications that can be propagated up the object hierarchy: a notification of changes to a file in a CassDAV server will result in notifications being sent to users monitoring the file (users who have carefully refined their subscription), and will also propagate the notification up to users monitoring any of the containing folders.

One new design principal for information sources is therefore to design the hierarchy of monitored objects to explicitly support both users who have time to carefully refine subscriptions by browsing deeply through object hierarchies, and to have high level, rapidly accessible objects for use in creating ad hoc subscriptions.

Consistency Across Related Information Sources

Subscriptions need to be generalizable across related information sources. For example, if a user subscribes to be notified of traffic problems while on one segment of a highway, there is a strong likelihood that when moving to a different segment of the freeway, the user will want to subscribe to the same or similar categories of information.

Support for this would require consistency across information sources that monitor the same types of information. For example, each traffic information source would have the same high level objects in its hierarchy, and only when you work your way down to monitoring certain on/off ramps do the object hierarchies of the different sources begin to look different.

Implementations of this (under the current CASSIUS architecture) would leave it to the awareness tool to

1) Note that two information sources are similar,
2) Determine that the user has subscribed to a certain set of information in the first information source,
3) Either automatically subscribe the user to similar information in the new information source, recommend it to the user, or make the information very easy to find and subscribe to [6].

An alternate approach would utilize the categorization and logging of awareness servers and information sources discussed in the prior section. It would allow users to look at a variety of related information sources and design a subscription that specifies what to do if information sources of that type are encountered in the future.

CONCLUSION

We have designed a set of guidelines for creating ubiquitous awareness environments, and provided an implementation of this environment. However, without strict guidelines in the design of information sources and awareness tool that work within this environment, the environment will only be usable for static subscriptions; subscriptions to information sources that will be a part of the user's life for an extended period of time. To make this environment usable for the creation of ad hoc subscriptions, information sources need to have both high level objects for rapid subscription and low level objects for refined subscription, sources of a common type need to utilize common object hierarchies, and the awareness tools need to be able to log, organize and recommend subscriptions based on information retrieved from the awareness servers that it encounters.

ACKNOWLEDGMENTS

This effort was sponsored by the National Aeronautics and Space Administration (NASA) Research Grant NAG2-1555. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NASA.

REFERENCES

1. Dourish, P., Bly, S. (1992) *Portholes: Supporting Awareness in a Distributed Work Group* In *CHI'92* ACM, pp. 541-547.
2. Gaver, W. W., R. B. Smith, and T. O'Shea (1991) Effective Sounds in Complex Systems: The ARKola Simulation In *CHI'91*, New Orleans.
3. Fitzpatrick, G., Mansfield, T., Arnold, D., Phelps, T., Segall B., Kaplan, S. (1999) Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin In ECSCW'99 Copenhagen.
4. Ishii, H., Wisneski, C., Brave, S., Dahley, A., Gorbet, M., Ullmer, B., Yarin, P. (1998) ambientROOM: integrating ambient media with architectural space In CHI'98 ACM, Los Angeles, pp. 173 - 174.
5. Kantor, M., Redmiles, D. Creating an Infrastructure for Ubiquitous Awareness, Eight IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001Tokyo, Japan), July 2001, pp. 431-438.
6. Maes, P. (1994) Agents that Reduce Work and Information Overload in *CACM,* **37,** 31-41.
7. Ramduny, D., Dix, A. and Rodden, T. (1998) Exploring the design space for notification servers In CSCW'98 ACM, Seattle, pp. 227-235.

A Web-based Infrastructure for Awareness based on Events

Roberto S. Silva Filho, Max Slabyak, David F. Redmiles
Information and Computer Science, University of California, Irvine
Irvine, CA 92697-3430 USA
+1 949 824 4121
{rsilvafi, mslabyak, redmiles}@ics.uci.edu

ABSTRACT
The ability to be aware of other people's work in a collaborative environment is essential to improving the coordination of the members of a group. In this context, events originating from many sources have to be filtered and combined in order to provide the right information to the right person at the right time. As the Web becomes a popular and ubiquitous way to integrate different information sources, an infrastructure that allows the filtering, combination, abstraction and routing of awareness information is required. In this paper, we describe DEAL (Distributed Event Awareness Language), an event-based infrastructure and language that supports the development of awareness applications in the context of distributed collaborative environments. The requirements of the infrastructure are discussed as well as the language syntax. Our motivation was to design a language and a set of usable and useful strategies and web services that provide usable and useful awareness capabilities for the development of awareness applications.

Keywords
Event Notification, Distributed Awareness, Event Processing Languages, CSCW, Web Services.

INTRODUCTION
In a broader sense, awareness refers to people's ability to sense relevant changes in their environment. In the specific context of CSCW, awareness is usually related to the perception of the direct or indirect changes in the computational and social environments originated by other people in the group. These changes are usually communicated through different computational resources, devices and applications. Awareness is an essential element in distributed collaborative environments. Individuals and groups of people need to be aware of what other members of the group are doing, in special, the activities of these other members that directly or indirectly affect each individual's work. For example, a manager needs to be aware of the progress of the tasks assigned to her subordinates, the presence of other members of the group in the workplace in order to see a demo, the arrival of a co-worker in a remote site, in order to start a chat session and so on. These activities can be modeled as events, atomic asynchronous messages that represent changes in the computational and social contexts.

The CSCW literature describes many ways to provide awareness in a collaborative and potentially distributed work environment [15]. Some examples include periodic pictures of a co-worker's office as provided by NYNEX Portholes [16], notifications about changes in shared artifacts, provided by the BSCW system [17], application monitoring gauges as described in [5] and so on. In order to integrate and combine the information coming from those different sources, allowing the processing, routing and filtering of such information, an event processing infrastructure is usually adopted [3].

Today's Web Services infrastructure focuses on providing a common set of protocols for the development of web applications, defining a web-based middleware for the development and integration of distributed web services [19]. In this infrastructure, the SOAP protocol is used as a remote procedure call mechanism for services communication; the UDDI implements a service location mechanism and the WSDL allows the description of the interfaces of the web services. This infrastructure alone, however, does not provide all the functionality necessary for the implementation of more sophisticated web applications [11]. In this context, event-processing services supplement this basic infrastructure providing the ability to integrate, process, select and route events originated from different nodes in the Web. In an event notification service, producers and consumers of information are separated by an event middleware that integrates these two parties, allowing different consumers to subscribe to information coming from many event producers. This event routing layer decouples event producers from their consumers, allowing the dynamic addition and removal of these components in the system.

Specifically, previous work analyzes the use of event notification servers in CSCW applications [4].

In this paper, we present DEAL (the Distributed Event Awareness Language), an event-notification infrastructure and language to support the development of distributed awareness applications for the Web. The DEAL environment extends the basic functionality provided by event notification servers such as Khronika [9], CASSIUS [8], CORBA Notification Service [12], ELVIN [6] and SIENA [1] to cope with the richer set of requirements of awareness applications. This is accomplished by the use of a powerful and usable event language that allows the definition, processing, combination, filtering and routing of events coming from heterogeneous sources (programs, applications, components, people, mobile devices and so on). The DEAL language syntax and resources were inspired in the features provided by event processing languages such as GEM [10], Yeast [2], EDEM [7] and READY [18]. Usability and simplicity with expressiveness were some of the principles considered in the design of the language. In special, the following scenario provides a set of requirements and motivations that guided the design of the language and infrastructure.

SCENARIO AND MOTIVATION

Consider an IT company with many branches in different cities over the country or even the world. Many people cooperate in different projects at the same time, performing different roles (manager, programmer, tester, designers and so on). Each project usually is carried on by a dynamic group of people whose interaction varies according to the group's current focus. At the beginning of a project, for example, designers and project managers are more active, whereas more towards the end, the activities concerning programmers, engineers and testers are more prominent. The work can also be split between different branches of the company. One branch, for example, deals with the product support while the other deals with the design and implementation. People join and leave groups as necessary. The group work is usually supported by different tools, such as configuration management repositories, word processing, CAD, databases and so on. These tools are used to produce and manage the evolution of the artifacts being developed, as well as their meta-information (documentation, specifications, metrics and so on). In this scenario, different people need to be aware of other group member's activities.

An indirect way of being aware of other people or group activities is to gauge the evolution of the artifacts they produce or modify. The kind of information one is interested in depends on her role in the organization. For example, programmers and engineers may want to be notified when some modules in a software project repository are ready for integration or when some change request is issued and consolidated in a defect report database. Managers, on the other hand, can gauge the activity in certain project by visualizing a graph with the number of changes in the project repository over the day.

In this dynamic work environment, mobility and heterogeneity is another concern. Computers are no longer limited to users' desktops at their workplaces; instead they are increasingly mobile and ubiquitous. Users can interact with a collection of computational devices ranging from non-stop servers, workstations, and motion sensors to portable devices as laptops, mobile phones and PDAs. In this mobile environment, awareness applications have to adjust their intrusiveness and information delivery policies to comply with different contexts (time, place, physical, organizational, administrative roles so on). For example, managers may want to be constantly informed about the progress of their projects using their portable computers. The level of attention required by the user, however, is dependent on her context. For example, one may want to be immediately notified about a meeting when she is in her office. This same information, however, may not be very important when she is at home or on a business trip.

Information persistency is another important issue. A manager, coming from a business trip, for example, may want to gauge the progress of a project by analyzing the event history of the preceding week. This requires having a way to store events during a certain period of time so they can be delivered when the user's computer is on-line again.

Not all events, however, should be stored for further analysis. People usually do not want to be notified about transitory and ordinary events. For example, the arrival of someone in her office or the temporary unavailability of a printer, that ran out of paper. This requires a mechanism to discard old events and filter irrelevant information.

Finally, meaningful events are usually a result of or are expressed as a combination of more ordinary events whose occurrence usually obeys some predefined patterns. For example, the turning on of the light of someone's office followed by the typing of some characters in the computer keyboard may indicate the arrival of this person at her workplace.

REQUIREMENTS FOR AWARENESS APPLICATIONS

The previous scenario illustrated many features that our event-notification infrastructure must support such as the integration of heterogeneous event sources, the need to compose events, the ability to filter information, events expiration time and mobile applications support. Moreover, the appropriate delivery of an event depends on the user context, timing constraints, roles and priorities. The notion of groups is also required.

Functional Requirements

To cope with these requirements, the DEAL event language and infrastructure was defined to provide the following features.

Subscriptions: Logical expressions that provide the ability to select a subset of events based on their content or type. They allow the routing of events to the right person at the right time, with the appropriate priorities based on the user context, group, role and other properties.

Abstraction: A mechanism that combines different events into higher-level notifications in order to provide more meaningful awareness information. Event abstraction can use the following strategies:

- **Pattern matching**: The ability to subscribe to event sequences and patters. It is the basis for abstraction and reduction. It may detect events in a specific order or out of order.
- **Reduction**: Translates sets of repetitive events, expressed as a pattern matching expression, into local state variables or higher-level events. This is specially required in monitoring applications, to prevent event flooding. A reduction consists in creating a new event indicating that an event pattern was detected.
- **Aggregation**: Is a more elaborated case of reduction in which the event generated is a composition of some of the attributes of the events in the detected pattern. Events are combined in a higher-level event which summarizes the content of the events in the pattern expression.

Event Condition Action (ECA) Rules: Special types of subscriptions that allow the execution of external applications or general actions whenever a logical condition is evaluated to true. For example: an action can add or remove subscriptions or even other rules; generate aggregated events, change policies, and invoke external applications in response to an event pattern detection. Rules can be used to evolve the behavior of the application in response to changes in the user environment.

Time constraints: Express Delivery intervals, time to live, as well as timing and temporal relations. Some events need to be detected within certain time interval in order to have some correlation. Transitory conditions may also be expressed by events with expiration time.

Subscription Priorities: Subscriptions are dependent on global, group and local contexts, allowing the adjustment of the information delivery according to the needs of the information consumers (or users).

Groups: Subscriptions and rules can be associated to groups, which are first class entities in DEAL language. This allows the broadcast of events, the definition of shared policies and contexts.

Hierarchical description of event sources: One of the neglected issues in some event infrastructures is the ability to answer the question "What can I subscribe to?" and "Which events are produced by each source?" The DEAL infrastructure provides the ability to browse through different event sources and to identify their events by keeping meta-information about which event sources are available and what events they produce.

Quality of Service Requirements

Apart from the main language features, the DEAL infrastructure allows the specification of different qualities of service and policies as follows.

Persistence of events and subscriptions: Events and subscriptions are persistent by default, allowing the support for mobile applications and pull delivery policy.

Mobility support: Clients are allowed to explicitly indicate their intent to move to a new location, allowing the infrastructure to perform the necessary migration operations such as the update event routing tables, the buffering of events or change the current qualities of service. This is performed by the **move-in**/**move-out** commands. Apart from these explicit commands, the infrastructure deals gracefully with the sudden disconnection of the event sources consumers.

Event Delivery Policies: During the specification of subscriptions and filters, one can specify which delivery policy to adopt, whether **pull** or **push**.

Security Policies: Authentication of groups, consumers and producers allow the event-processing infrastructure to prevent unauthenticated clients from receiving unauthorized events.

THE EVENT LANGUAGE

This section describes the DEAL event language. Examples are presented to illustrate its use and syntax.

Logical Expression Operators: >, <, >=, <=, == as well as `starts_with` and `ends_with`.

- `MyType:ev1.name` **`starts_with`** `"Ro"`

Subscriptions: Defined using the **subscribe** keyword, and removed by the **unsubscribe** command. Whenever a subscription is evaluated to true, a notification is produced having the list of events used in the subscription expression.

- **`subscribe`** `mySub SomeType:ev1.name == "Mike"` **`and`** `OtherType:ev2.counter == 2`
- **`unsubscribe`** `mySub`

Event Type Definition: Creates an event type, a structure supporting: boolean, string, long, int, as well as other Java basic types

- **`type`** `Type1 {name:` **`string`**`, age:` **`int`**`, is_present:` **`boolean`**`}`

On-the-fly event declaration and instantiation:

- `Type1:ev1 = {"john", 22, true}`

ECA Rules: Described by the use of the keyword **rule,** which uses the **do** command to define the action to be executed when the rule is matched.

- `rule myRule ev1.name == "check-in" and Local.time > 12pm do run myApplication.exe`

In this example, the **run** is a reserved word that allows the execution of external applications.

```
rule otherRule ev1.name == "turn-on-light"
and ev2.name = "workstation-activated" do {
type MyAbstrac = {name: String};
MyAbstrac ev3;
ev3.name = ev1.name + ev2.name;
notify ev3; }
```

Rule Activation: Activates and deactivates rules according to the context using the **enable** or **disable** commands.

- `rule activateRule1 Local.time > tomorrow do enable rule1`
- `rule deactRules ev1.description = "end of meeting" do disable rule1 rule2 rule3`

Temporal Expressions: Express time constrains between events. Time triggers and ranges: **at, by, in, within.**

- `at 10pm` - matches after the next occurrence of 10 pm
- `by 10pm` - matches from now until the next occurrence of 10 pm
- `in 2 hours and 10 minutes` – matches after 2 hours and 10 minutes from now
- `within 3 hours` – matches permanently in a period between now and 3 hours ahead

Time Period expressions: **today**, **daily**, **weekly**, **monthly**, **yearly:**

- `at 10 am daily`
- `at monday weekly`

Event Validity Check (time to "live"): Is a special attribute, present in all events, which expresses its expiration date and time. The expiration condition can be evaluated using the **expired** keyword:

- `expired ev1`
- `ev2.expiration < today and ev3.expiration < tomorrow`

Pattern Matching: Performs the matching of a sequence of events (repetition, sequence and optional).

- Enforced order: `ev1 then ev2 then ev3`
- Optional order: `ev1 and ev2 and ev3`
- Matching of repetition of events (0 or more and 1 or more): `repeat (ev1 and ev2) 2 times`

Groups: The keyword **group** allows the definition of sets of users, the keywords **add** and **remove** perform the addition and removal of users to a group, whereas the operand **in** checks the pertinence of users in groups.

- `group g1 { user1, user2, user3}`
- `add g1 user4 // adds sub4 to group g1`
- `remove g1 user2`
- `ungroup g3 // removes the group`
- `user1 in g1`

Groups can be used as parameters of the **notify** command to broadcast events, as in the example

- `rule r1 ev1 then ev2 then ev3 do notify ev1 to g1`

Roles: A role is a group of users. Groups are used to represent roles. This allows users to perform different roles.

Contexts: Contexts are name spaces that define scopes where some properties, rules and subscriptions are valid. Local and global variables (or properties) can be stored in the local, group or global contexts. In addition, the contexts provide a set of predefined environment variables that allow the access to information as local time, hostname, user name and so on. The contexts are accessed through the special types **Local** and **Global.**

- `at 12 pm and Local.mycounter == 12`
- `at 12 pm and Global.members > 5`

Local and global rules can be defined. Expressions can use values of these contexts. A rule is associated to the local context scope using the **local** modifier. Global rules can be defined using the **global** modifier.

- `local rule myRule at 12 pm and mycounter == 12 do enable rule1`

In this example, *mycounter* is a variable in the Local scope. Each group has a special context associated with it. Group contexts are accessed by the group name, for example: g1.size expresses the size of the group.

Group rules can be defined using the **group** modifier followed by the group name, before the rule declaration.

Attributes can be added or removed from a context using the **addcontext** and **remcontext** keywords.

- `addcontext Local name:string`

Events and Subscription Priorities: Special attributes in the events, which are used by the system to perform event routing. They can be used in subscriptions by accessing the reserved attribute **priority**.

- `rule adjustPriority Local.time > 6 pm do ev1.priority = ev1.priority –1`

DESIGN

The DEAL architecture is described in Figure 1. The system is implemented as a wrapper around a notification server infrastructure such as CASSIUS [8] or Siena [1].

The event-processing kernel implements the DEAL functionality using the resources provided by the event notification server. Applications interact with the infrastructure through a programmatic API while end-users can use a command interpreter shell or a more sophisticated GUI. The interaction with the system can be intermediated by Web services interfaces such as SOAP or by lower-level protocols as HTTP/CGI. The architecture of the system can be distributed, using the resources of federated notification servers, according to the features provided by these systems. In special, Siena provides a scalable web-based content-routing infrastructure.

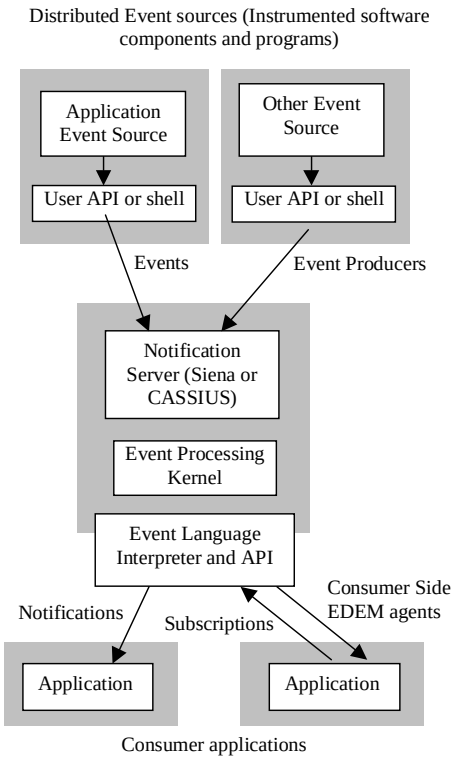


Figure 1 Design of the DEAL infrastructure using an event notification server

IMPLEMENTATION

DEAL is being implemented using the Java (J2SDK1.4) programming language and the CASSIUS notification server. CASSIUS was chosen for its ability to manage and provide access to a hierarchical list of event sources and their associated events. It also provides a subscription editor GUI that facilitates the interaction with the end users. For using the HTTP/CGI protocol, CASSIUS allows DEAL to be integrated with different event sources distributed over the Web, providing an event-based infrastructure for awareness information.

RELATED WORK

In this section, we summarize the main systems that inspired the DEAL language.

EDEM

EDEM (Expectation-Driven Event Monitoring) [7] is a user interface validation and monitoring tool. EDEM uses agents to monitor GUI event patters according to design use expectations. The agent description language is very complete and allows the detection of event patters, the manipulation of local context (in the monitored site), the definition of higher-level events (abstraction), the collection of repetitive events (reduction) and so on.

The EDEM architecture is defined in order to collect usability data. Since agents execute together with the application being monitored, the system was not designed to monitor events coming from multiple distributed applications.

Yeast

The Yeast (Yet another Event-Action Specification Tool) [2] is an event-action system used to automate tasks in a UNIX environment. Yeast allows actions to be performed when event patterns and environment changes are detected. It allows the association of temporal constraints to events, borrowing its syntax from the *at* and *cron* programs of UNIX systems. Sequential and out or order event pattern detection is supported. User-defined actions are executed whenever an event pattern match occurs. These actions can originate new events or start different applications. Yeast allows the definition of rules to be defined, activated or deactivated at runtime. This flexibility is provided by a shell script interface that integrates the UNIX shell commands with yeast pre-defined keywords.

The system is very complete, providing many features that can be used to support the development of awareness applications. It, however, was developed to operate on UNIX environments, being limited at monitoring its specific resources and objects such as processes, files and user logs. Users can define their own events but their types are not enforced. There is no advertisement of the event types provided by an event source. Event sources are not primary entities in this model. There is no explicit idea of subscription and subscriber. The event language does not allow the creation and manipulation of local variables, limiting the support for local and global contexts. User groups are not supported.

Khronika

The Khronika [9] is a centralized event notification service created to increase people awareness about their environment. One of the objectives of the system is to bridge the gap between computational and real-world events. Each user of the system can specify sets of pattern-action subscriptions that are used to automatically notify the user when an event pattern is detected. Khronika also allows the direct browsing of the events in the repository. Events have expiration time and remain on the server database as specified in their validity (days, hours or brief intervals). The event language allows queries by time interval, event

types and substring matching. Similar to Yeast, there is a mapping between English expressions as "today", "tomorrow", "now", "Thursday afternoon", and so on, to more precise time constraints. Access control lists and user groups are used. These restrictions are made simple for usability purposes.

Khronika does not provide the ability of abstracting and aggregating events. There is no support for different event sources, including mobile devices as well as the ability to activate/deactivate subscriptions (or rules) based on environment changes. There is no notion of user groups.

Gem

GEM [10] is a generalized event language for real-time distributed systems monitoring. It allows the event sequence detection and the specification of rules that can be activated or deactivated according to other rules. For being designed for real-time monitoring, rules can include special time constraints concerning incoming events delays. It also allows the use of event order constraints in event expressions, such as the specific order events should occur and the acceptable delay between them. Events can be abstracted and generated based on contents of other events. There is support for abstraction.

The GEM language itself was not defined for usability. It does not provide support for context and groups.

READY and CORBA

READY (Reliable Available Distributed Yeast) [18] is a general-purpose event notification service based on YEAST. READY adds to YEAST the ability to handle compound event matching, quality of service and other event constructs, in an implementation that extends the Standard CORBA notification server [12].

In its porting to CORBA [13], READY lost the simplicity, elegance and easy-to-use interface of the Yeast model. Its language became more complicated, being based on the OMG Event Notification Language. It also lost the timing constraints neutrality and elegance of Yeast.

CONCLUSIONS

DEAL is an event processing language and system designed to provide awareness information in a heterogeneous distributed system. The system was designed to cope with current distributed systems characteristics as mobility, heterogeneity, timing, as well as CSCW aspects as groups, context and priorities. In order to do so, it combines characteristics of different event processing, monitoring systems and awareness driven notification servers. It was specially designed as a distributed awareness service that can be integrated in many Web applications. This is accomplished by the use of the HTTP protocol. The event language was designed to be useful and usable, providing a high-level way to interact with the system. A prototype is being implemented in Java using CASSIUS as the basic notification service.

ACKNOWLEDGMENTS

This effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599; by the National Aeronautics and Space Administration (NASA) under contract NAG2-1555; by the National Science Foundation under grants 0083099 and 0205724. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force Laboratory, or the U.S. government.

REFERENCES

1. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems, 19(3):332-383, Aug 2001.
2. B. Krishnamurthy and D. S. Rosenblum. Yeast: a general purpose event-action system. IEEE Transactions on Software Engineering, Vol. 21, No. 10. October 1995.
3. D. C. Luckham. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise System. Pearson Education. ISBN 0-201-72789-7. Boston MA 2002.
4. D. Ramduny, A. Dix and T. Rodden. Exploring the design space for notification servers. Proc. of the ACM CSCW'98. pp. 227-235. Seattle, 1998
5. De Souza, C.R.B., Basaveswara, S. D., Kantor, M. Redmiles, D.F. Lessons Learned using Event Notification Servers to Support Awareness. Human Computer Interaction Consortiun'02- Winter Workshop, Jan 31 to Feb 3, Fraser, CO. 2002
6. Fitzpatrick, G., Mansfield, T., et al. Augmenting the Workaday World with Elvin, Proceedings of 6th European Conference on Computer Supported Cooperative Work (ECSCW 99), Kluwer, 1999, pp. 431-450.
7. Hilbert, D., Redmiles, D. An Approach to Large-Scale Collection of Application Usage Data Over the Internet, Proceedings of the Twentieth International Conference on Software Engineering (ICSE '98, Kyoto, Japan), IEEE Computer Society Press, April 19-25, 1998, pp. 136-145.
8. Kantor, M., Redmiles, D. Creating an Infrastructure for Ubiquitous Awareness, Eight IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001 Tokyo, Japan), July 2001, pp. 431-438.
9. L. Lovstrand. Being selectively aware with the Khronika System. Proc of ECSCW'91. 1991.

10. M. Mansouri-Samani and M. Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems. In ICODP/ICDP'97. 1997.

11. M. Stal Web services: beyond component-based computing. CACM Special Issue: Developing and integrating enterprise components and services .Vol. 45, Issue 10. pp. 71-76. October 2002

12. Object Management Group. Notification Service Specification v1.0.1. August - 2002. http://cgi.omg.org/docs/formal/02-08-04.pdf

13. OMG Common Object Request Broker Architecture – (CORBA/IIOP) v.3.0. formal/2002-06-33. http://cgi.omg.org/docs/formal/02-06-33.pdf

14. P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. ACM Transactions on Computer Systems. Vol 13. Issue 1. Feb 1995.

15. P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. ACM Proc. of Conference on CSCW'92. Toronto, Ontario, pp. 107-114. 1992

16. P. Dourish, and S. Bly. Portholes: Supporting awareness in a distributed work group, in Proceedings CHI'92, Monterey, CA, ACM/SIGCHI, 1992.

17. R. Bentley, W. Appelt, U. Busbach, E. Hinrichs, D. Kerr, K. Sikkel, J. Trevor and G. Woetzel. Basic Support for Cooperative Work on the World Wide Web. International Journal of Human Computer Studies 46, pp. 827-846, 1997.

18. R. E. Gruber, B. Krishnamurthy, and E. Panagos. High-level constructs in the READY event notification system. In 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications, Sintra, Portugal, September 1998.

19. Web Services Activity. 2002 http://www.w3.org/2002/ws/

Management of Interdependencies in Collaborative Software Development

Cleidson R. B. de Souza[1,2] David Redmiles[1] Gloria Mark[1] John Penix[3] Maarten Sierhuis[4]

[1]*School of Information and Computer Science, University of California, Irvine*
Irvine, CA, USA

[2]*Departmento de Informática, Universidade Federal do Pará*
Belém, PA, Brasil

[3]*Computacional Sciences Division, NASA/Ames Research Center*
Moffett Field, CA, USA

[4]*Research Institute for Advanced Computer Science, NASA/ Ames Research Center*
Moffett Field, CA, USA

Abstract

In this paper we report results of an informal field study of a software development team conducted during an eight week internship at the NASA/Ames Research Center. The team develops a suite of tools called MVP, and is composed of 31 co-located software engineers, who design, test, document, and maintain the different MVP tools. We describe the formal and informal approaches used by this group to manage the interdependencies that occur during the software development process. Formal approaches are legitimated by the organization, whereas informal approaches emerge due to the needs of the developers. We also describe how the software development tools used by this team support these approaches and explore where explicit support is needed. Finally, based on our findings, we discuss implications for software engineering research.

1. Introduction

Software development is typically a collaborative activity in which experts from different domains work together to produce a software artifact. Indeed, formal and informal communication account for more than half of developers' time [21], and cooperative activities account for about 70% of this time [30]. Therefore, breakdowns in communication and coordination efforts constitute one major problem in software development [3].

One of the reasons that cooperative software development is difficult is the large number of interdependencies that occur. These include interdependencies among activities in the software development process, among different software artifacts, and finally, in different parts of the same artifact. One example involves the design document and the requirements specification—if the specification changes, the design normally needs to be changed as well. Another example involves dependencies among parts of the same artifact, such as program dependencies—syntactic relationships between the statements of a program that represent aspects of the program's control flow and data flow [22].

Software engineering has already identified the need to manage these interdependencies and has been developing formal approaches to deal with them. For example, software development processes describe, among other things, when each artifact should be created during the software development effort. Such processes would prescribe that the requirements specification to be created before the design document to minimize problems due to the dependency between these documents. Design techniques have also been developed. Examples of such techniques include information hiding [19], which tries to minimize dependencies in the implementation by using the concept of coupling, and design patterns [7], which give dynamic (runtime) program dependencies explicit representation as static program structures, making them easier to manage. In addition to formal approaches, software engineering tools have been built to support the management of interdependencies. An example is configuration management systems that deal with dependencies in the source code.

Informal approaches are also used to manage the interdependencies. These practices exist because no matter how formal and well-defined a process may seem, it will always be incomplete, and also because formal approaches have practical limitations [8]. Informal approaches are as important as formal approaches and need to be understood if one wants to provide support for software development. Informal approaches solve problems not addressed by formal approaches, so formal and informal approaches complement each other. An example of an informal approach is the use of formal communication channels in software development organizations to deal with dependencies among components of the same subsystem when the developers are co-located [9].

In this paper, we describe an informal field study that analyzes both formal and informal approaches used by a software development team to manage the interdependencies that occur during software development. We classify this work as an informal study since it consists primarily of observations made by the first author during an eight-week internship during the Summer of 2002. The formal

approaches identified here are those legitimately adopted by the organization, such as the software development process; the software development tools used, namely the configuration management (CM) and bug-tracking tools; and other approaches, such as the division of labor, formal meetings, and so on. The informal approaches are the emerging practices adopted by the team to deal with these interdependencies, such as the adoption of conventions; partial check-ins; problem reports (PRs) that cross work boundaries; and the role of e-mail as a coordination mechanism. Our observations build on Grinter's work [9]; we identify several other informal approaches and analyzed the role of formal approaches in the management of interdependencies. The identification, analysis, and support for formal and informal approaches are essential in improving software development efforts. Interdependencies affect the coordination success because they decrease the certainty of a project [13].

2. The MVP Software Development Team

The field study was conducted in cooperation with a team that develops a software application, which for the purposes of this paper we call MVP (All names were changed to preserve anonymity). MVP is a suite of 10 different tools developed at NASA/Ames Research Center. The MVP source code is approximately one million lines of C and C++.

2.1. Team Organization

The MVP team is divided in two groups: developers and V&V staff. Developers are responsible for writing new code, fixing bugs, adding new features, and so on. This group comprises 25 members, 3 of whom are also researchers who write their own code to explore new ideas. The overall experience of these developers ranges from 3 months to more than 25 years. Experience in the MVP group ranges from 2.5 months to 9 years. This group is spread along several offices across two floors in the same building.

V&V members are responsible for testing and reporting identified bugs, keeping a running version of the software for demonstration purposes, and maintaining the documentation (mainly user manuals) of the software. This group comprises 6 members, half located in the V&V Laboratory, and the rest in several offices on the same floor as the laboratory. The V&V Lab and the developers' offices are located in the same building.

2.2. The MVP Software

Each of the MVP's 10 tools uses a specific set of "processes." A process, for the MVP team, is a program that runs with the appropriate run-time options. It is not formally related to the concept of processes in operating systems and/or distributed systems. MVP's processes typically run on distributed Sun workstations and communicate using a TCP/IP socket protocol. Running a MVP tool means running the processes required by this tool with their appropriate run-time options. Processes are used to divide the work among the developers (see section 4.3).

3. Methods

As an intern with the MVP team, the first author was able to make observations and collect information about several aspects of the team. Additional material was collected by reading manuals for the MVP tools, manuals for the software development tools used, formal documents (such as the description of the software development process and the ISO 9001 procedures), training documentation for new developers, problem reports, and so on, as well as talking to colleagues. Some of the team members—the documentation expert, V&V members, testers, process leaders, and process developers—agreed to let the intern shadow them for a few days to better learn about their functions and responsibilities. A representative subset of the MVP group was interviewed. Interviews lasted between 45 to 120 minutes. A total of seven interviews [15] were used to find out about the usage patterns of various tools. The data has been analyzed by using grounded theory [28].

4. Formal Approaches

Formal approaches are those legitimately adopted by the team to support the management of interdependencies. They facilitate the software development effort by improving the coordination of activities. These approaches have long been studied in the software engineering and organizational research literature (e.g., [6, 26]), so we will mention only aspects of these approaches in the context of the MVP team.

4.1. The Software Development Process

The MVP team uses a formal software development process that prescribes the steps needed to be performed by the developers. For example, the following steps must be performed by all developers after finishing the implementation of a change. Initially, they should integrate their code with the main baseline. After that, must test their changes to check if their integrations have inserted bugs in the code. Finally, after checking-in files into the repository, developers must send e-mails to the software development mailing list describing the problem report (PR) associated with the changes, the files that were

changed, and the branch where the check-in will be performed, among other pieces of information.

The MVP software process also prescribes the usage of code reviews before the integration of any change and design reviews for major changes in the software. Code reviews are performed by the manager of each process. Therefore, if a change involves two processes, a developer's code will be reviewed twice: once by each manager. Design reviews are recommended for changes that involve major reorganizations of the source code; their use is decided by the software manager.

4.2. The CM and Bug Tracking Tools

We observed that MVP developers employ mainly two software development tools for *coordinating* their work: a configuration management (CM) system and a bug-tracking system [2, 9, 11]. These tools are integrated so that there is a link between the PRs (in the bug-tracking system) and the respective changes in the source code (in the CM tool). Both tools are provided by one of the leader vendors in the market. Other tools, such as CASE tools, compilers, linkers, debuggers, and source-code editors, are also used.

A CM tool supports the management of source-code dependencies through its embedded building mechanisms, which indicate what parts of the code need to be recompiled when one file is modified. In this case, we use Grinter's classification of dependencies: "Compile-time dependencies occur when a sub-system is being compiled. Build-time dependencies occur when several subsystems or the entire system is being linked. Run-time dependencies occur when the executable is running [9]." According to this classification, CM tools support compile and build-time dependencies. Similarly, a bug-tracking tool, when associated with the CM tool, supports the tracking of changes performed in the source code during the development effort.

Two members of the MVP team play important roles in the usage of these tools: the configuration and release manager and the bug-tracking manager. Both help in the administration of the tools and try to relieve the developers of some of most common tasks (e.g., the CM manager created a command interface on top of the CM tool to make it easier for MVP developers to use). The CM manager provides full-time support for the CM tool, and the bug-tracking manager is also an MVP software developer. Both managers have been receiving training in those tools, and other developers are trained before starting work in the group. Their training includes the software development tools and the MVP software development process.

The MVP team employs several advanced features of the CM tool, such as triggers, "winking in" techniques to reduce compilation time, labeling, and branching strategies. Indeed, the branching strategy employed is one of the most important aspects of a CM tool because it principally affects the work of MVP developers. It is a way of deciding when and why to branch. This strategy affects the task of coordinating parallel changes. According to the nomenclature proposed by Walrad and Strom [31], the following branching strategies are used by the MVP team: (1) branch-by-purpose, in which all bug fixes, enhancements, and other changes in the code are implemented on separated branches; (2) branch-by-project, in which branches are created for some of the development projects; and (3) branch-by-release, in which the code branches upon a decision to release a new version of the product. The branch-by-purpose strategy is employed by MVP developers in their daily work, whereas the other strategies are used only by the CM manager. In other words, the developers themselves create new branches for each new bug fix or enhancement, but branches for projects and releases are created only by the manager.

The branch-by-purpose strategy supports a high-level of parallel development by allowing developers to work on different branches at the same time, thus avoiding problems that exist in other strategies [31]. According to this strategy, each developer is responsible for integrating his or her changes into the main code, which is often called "push integration" [1]. The changes are then available to all other developers. Therefore, if one bug is introduced, other developers will notice it because their work will be disrupted. Indeed, we observed and collected reports of different instances of this situation. A developer who suspects there is a problem introduced by recent changes will contact the author of the changes to check the change, or to provide more information about it.

4.3. Other Approaches: Meetings and Division of Labor

MVP developers employ other formal approaches to manage the interdependencies in the software. For example, the V&V group holds weekly meetings to discuss problems, deadlines, etc. These meetings are also used for official announcements, such as trips, dates of new releases, demonstrations, audits, and so on. Likewise, the entire MVP team (developers and V&V staff) holds bi-weekly "*software pre-design meetings.*" In these meetings, formal announcements are also made, but the most important part of the meeting involves the discussion of new PRs. In this case, the developers each announce their new PRs, describing them through their number and headline. In general, the headline provides enough information about the nature of the PR, but other developers might ask for more details. This is an opportunity for developers to discuss their work, obtain help, and be aware of what is happening in the team. For example, it is not

uncommon after a developer reports a PR that another developer mentions that the problem has already been fixed. PRs that are almost finished might also be announced to warn others about possible "weird" behavior in the tools. Finally, during these meetings the software manager will decide if design reviews are necessary.

The MVP software development team also adopts a clear division of labor based on the processes that compose each MVP tool. Each developer is assigned to one or more processes and tends to specialize in it. There are process leaders and process developers, who mostly work only on a particular process. This is important because it allows the developers to understand the behavior of the process more deeply and become familiar with its structure, therefore helping them to deal with the complexity of the code. Indeed, during the software development activity, managers tend to assign work according to these processes. However, it is not unusual to find developers working in different processes under various circumstances (e.g., before launching a new release, a developer might be assigned to fix bugs in other processes). Developers also work in different processes due to the continuity of the work. Sometimes bugs that seem to be located in a process and therefore are allocated to the developer who works with this process are later discovered to be located in another process. In this case, it is better to let the developers finish the work because so much time was invested in it. Thus, this allows developers to gain a comprehensive view of the whole MVP software.

5. Informal Approaches

Informal approaches are the practices adopted by the MVP team to deal with the interdependencies that occur during the software development process. We call them informal because they emerged naturally in response to the needs of the team and are not taught to new members. The approaches that we identified are discussed below.

5.1. Problem Reports Are Boundary Objects

In our analysis we identified that PRs are used to facilitate the management of interdependencies of developers from different groups and with different roles. In other words, PRs are "boundary objects" in the sense of Star and Griesemer [27]: objects whose common identity is robust enough to support coordination, but whose internal structure, meaning, and consequences emerge from local negotiations between groups. Indeed, PRs are used by end-user liaisons, developers, and testers for different purposes.

Consider the following. When a bug is identified, it is associated with a specific PR. Whoever identified the problem is also responsible for including information about 'how to repeat it' in the PR. This description is used by the developer assigned to fix the bug to specify the circumstances (adaptation data, tools, and their parameters) under which the bug appears. After fixing the bug, this developer must fill a field in the PR that describes how the testing should be performed to properly validate the fix. This field is called 'how to test.' This information is then used by the test manager, who creates test matrices that will be used later by the testers during regression testing. The developer who fixes the bug also indicates in another field of the PR whether the documentation of the tool needs to be updated. Then, the documentation expert uses this information to determine whether the manuals need to be updated based on the changes the PR introduced. Finally, another field in the PR conveys what needs to be checked by the manager when closing it. Therefore, the PR reminds the software manager of the aspects that need to be validated.

In short, the information provided by the PR is used by the developers to manage the several interdependencies in the software being developed. For example, since the user manual of an MVP tool depends on part of that tool's source code, so changes in this source code need to be reflected in the manual. The information about such changes is provided to the documentation expert through one of the fields in the PR.

5.2. Naming Conventions

Developers share repositories containing the source code (the CM tool) and information about changes in this code (the bug-tracking tool). As a result, the team establishes naming conventions that must be followed when dealing with these tools. Conventions are common and accessible rules or arrangements established in the group that act as a means to merge the different perspectives and work styles involved in handling shared objects [14].

An example of a convention is the naming convention used in the creation of branches in the CM tool: it must be based on the PR number recorded in the bug-tracking tool as well as on the developer's name. This allows the relationship that exists between a change and its corresponding PR to be clearly represented, therefore facilitating identification by MVP developers. However, these conventions are not properly supported by these tools, which is a source of complaints by the developers. Indeed, creating and naming branches is a cumbersome task with four or five different tedious steps that could be automated because they follow a naming convention.

5.3. E-mail Conventions

As mentioned before, the MVP software development process prescribes that after checking-in code into the repository, a developer needs to send an e-mail to the software developers' mailing list. However, we found out

that MVP developers perform these activities in the reverse order—they will send e-mail before, not after, the check-in. By doing so, MVP developers allow their colleagues to prepare for the changes. Indeed, developers might even send e-mail to the author of the change asking for a delay of its check-in. We also found out that in this same e-mail developers describe the impact that their changes will have on others' work. A developer who reads these e-mails might walk to the co-worker's office to ask about the changes or, if the change has already been committed, browse the CM and bug-tracking systems to understand them. The following list presents some usual comments sent by MVP developers:

"No one should notice."
"(…) only EDP users will notice any change."
"Will be removing the following [x] *file. No effect on recompiling."*
"Also, if you recompile your views today you will need to start your own [z] *daemon to run with live data."*
"The changes only affect [y]*-mode so you shouldn't notice anything."*
"If you are planning on recompiling your view this evening ([current date]) and running an MVP tool with live [z] *data, you will need to run your own* [z] *daemon."*

Sending e-mail before the check-ins with the description of the impact of the changes is an important convention because it allows other developers to prepare and reflect about the effect of their colleagues' changes in their current work. Because they are aware of some of the interdependencies in the source-code, they might consequently adjust to these changes.

In addition to the flexibility that allows the description of the impact of the changes, e-mail provides asynchronous communication, which requires storage of the messages until their delivery to the recipient. This is used by MVP developers to learn about what changed in the code in a certain timeframe. For example, these e-mails were used by a developer to catch up with the changes that occurred while out of the office. They contained information that allowed the developer to identify changes that did not affect current work, but might affect future work. The following comment from another MVP developer supports this:

"(…) all of the sudden you were working and everything was going great and an e-mail comes through, you look at it, it does not mean a lot, you blow it (…) you keep working and one hour later things were broken. Why is that not working? Oh, that last check-in! You go back to that e-mail: who did this? And maybe you can go talk to that person: 'you broke something' (…)"

The information in the e-mail is also important because it informs (or reminds) developers that they have been engaged in parallel development. Often, developers are unaware of parallel activity because they do not check the version tree that displays information about other developers working on the same file. The information in the e-mail is usually enough to tell the developer whether these changes should be incorporated right away or whether they can wait until just before check-in. In either case, the latest changes must be "merged back" into the developer's version of the file. In general, if one file has been checked-in several times and a developer has the same file checked-out, he or she "merges back" the changes indicated in the e-mail to avoid working with an outdated file.

The asynchronous nature of e-mail could be problematic because developers might miss important notifications about changes. However, during the field work, we did not notice any such problems. Furthermore, sending e-mail before a check-in is also used by other developers to support expertise identification and as a learning mechanism. Developers associate the author of the change with the "process" where the changes are being performed. In other words, MVP developers assume that if one developer constantly and repeatedly performs check-in in a specific process, it is very likely that the developer is an expert on that process. Therefore, another developer needing help with that process will look to that developer for help:

"[talking about a bug in a process that he is not expert] *(…) I don't understand why this behaves the way it does. But, most of these PR's seem to have John's name on it. So you go around to see John. So by just by reading the* [PR] *headline of who does what, you kind of get the feeling of who's working on what (…).So they* [e-mails] *tend to be helpful in that aspect as well. If you've been around for ten years, you don't care, you already know that* [who works with what], *but if you've been here for two years that stuff can really make difference (…)"*

In addition, the simple fact that developers read the e-mails sent by other developers to check for the impact of others' changes facilitates learning about the MVP software. Interestingly, the two developers who reported these aspects of e-mail were relative novices at MVP, with 2 years and 2.5 months experience there.

5.4. Holding onto Check-ins

As mentioned, MVP developers add to the e-mail the description of the impact of their changes in other developers' code. The two most common types of impact statements are changes in run-time parameters of a process and the need to recompile parts or the whole source

code[1]. The former case is very important because other developers might be running the process that will be changed. The latter case is described because when a file is modified, it, as well as the other files that depend on it, will be recompiled, and this recompilation process is time-consuming—up to 45 minutes. Developers are aware of the delay they might cause to others; therefore, they hold check-ins until the evening. According to one of the developers:

> *"(…) and the other thing that you find is that when people also know that if they are going to check-in a file they will do in the later afternoon … you're gonna do a check-in and this is gonna cause anybody who recompiles that day have to watch their computer for 45 minutes (…) and most of the time, you're gonna see this coming at 2 or 3 in the afternoon, you don't see folks (….) you don't see people doing* [file 1] *or* [file 2] *checking-in at 8 in the morning, because everybody all day is gonna sit and recompile."*

Holding onto check-ins is an informal approach adopted by the MVP software development team to minimize the problems caused by the interdependencies that exist on the source code. However, this is possible only because MVP developers are aware of the existing interdependencies.

5.5. Engagement in Parallel Development: Partial Check-ins and "Speeding Up" the Process

We also noted that MVP developers engage very often in parallel development. This happens when more than one developer has the same file checked-out. Conflicts might occur when one of these developers checks-in this file back into the repository because the other developer's version will then be outdated, and any changes that developer makes will potentially be inappropriate. To update the version, the developer needs to merge the other's changes back in his or her code. This operation is called by the developers "back merging," and in CM terminology is named "synchronization of workspaces." Due to the need to perform these back merges, a new dependency between artifacts is created during parallel development. This dependency occurs between any version of a file that has not yet been checked-in and the new version of this same file created after the check-in (i.e., the current version of a file checked-out by a developer is now dependent on the new version checked-in into the repository because the former needs to incorporate the changes of the latter before being checked-in). This is another example of dependency in software development.

Conflicting changes are more likely to occur in files that are accessed by several developers at the same time. For example, in MVP software, some files are used to define programming language structures that are used all over the code. Different developers often change these files, which means that they have a high degree of parallel development. These files are especially important because there is a significant correlation between them and the number of defects reported [20]. MVP developers reported that they do not avoid parallel development in these files because conflicts are infrequent and not likely to occur. But, without access to the CM tool, it was not possible to statistically test this claim. MVP developers accepted parallel development because it was necessary to achieve high productivity. However, we identified that they adopted a strategy to deal with files with a high degree of parallel development. To minimize the possibility of conflicts, developers would perform "partial check-ins," which consists of checking-in some of the files back into the repository, even when the developers have not yet finished all their changes. This strategy decreases the number of dependencies that occur, and consequently reduces the number of necessary back merges. Note that partial check-ins are variations of the formal software development process, which establishes that check-ins will be performed only when the changes in all files are finished.

Finally, according to Grinter [9], software developers might rush to finish their work when they engage in parallel development because they want to avoid merging. We identified that developers will rush only when they are testing their changes right before check-in. As one developer plainly pointed out: "This is a race!" According to the software development process, this testing is necessary to guarantee that the changes will not introduce bugs into the system. We observed that this testing is very informal. For example, developers will sit in the V&V Laboratory and compare the current version of the MVP with the one with changes. In short, MVP developers do not use regression testing at this moment. That will be used by the V&V staff before creating a new release of the software. This means that techniques that minimize the number of test cases necessary to validate the changes in the software (e.g., [23]) cannot be used by MVP developers to determine whether the tests they need to run can be impacted by changes that another developer makes. These techniques can be used only by the V&V staff.

Although we observed that some check-ins introduced errors, we do not have evidence that these errors were introduced due to this "racing." Similar to partial check-ins, "speeding up" the process is employed by the MVP developers to avoid the additional work necessary to deal with the extra-dependencies that occur during parallel development.

---

[1] The CM tool used by the MVP team allows developers to choose if they want to incorporate others' changes, meaning that they are able to decide if they want to recompile the code or not.

6. Computational Support for Informal Approaches

Figure 1 summarizes the formal and informal approaches used by the MVP team to manage the interdependencies that occur during their software development activities. As mentioned before, formal and informal approaches complement each other, so problems not solved by the formal approaches might be solved by the informal ones. For example, none of the formal approaches used by the MVP team addresses the issue of how to manage the crossing-boundaries dependencies that occur when a change is committed into the repository. This problem is solved by the MVP team by adopting a particular PR structure that provides information for developers with different roles (see section 5.1).

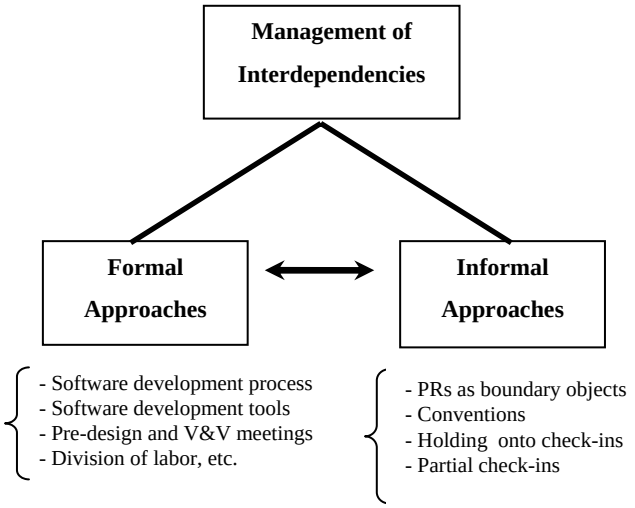


Figure 1: Formal and Informal Approaches Adopted by the MVP Software Development Team

The tools used by the MVP team assist some of the informal approaches. For example, the CM tool allows software developers to perform partial check-ins. In contrast, due to the lack of tool support, developers need to rush to finish their work when they are testing their changes. In this section, we discuss the existence (or lack) of support for informal approaches in more detail. In addition, we discuss implications for software engineering research when there is a lack of support.

6.1. Problem Reports as Boundary Objects

Bug-tracking tools are flexible enough to allow their managers to define the fields that will compose a PR. In addition, these tools allow a manager to specify a simple workflow describing *when* each one of these fields needs to be filled in [12]. By doing that, they allow the creation of PRs with fields that contain information that is useful to developers who are members of different groups. In the MVP team, the information in these fields describes how each developer's work is going to be affected by the PR. This means that these tools allow PRs to be defined and used as coordination mechanisms to manage interdependencies during software development.

6.2. Support for Naming Conventions

Following conventions for dealing with shared objects (or repositories) implies additional effort; hence, technical support often is needed [14]. As mentioned before, MVP developers follow a naming convention in which the name of the branches in the CM tool should be based on the PR number recorded in the bug-tracking tool. MVP developers have complained that the task of creating branches is very cumbersome, with four or five different tedious steps to be performed. Because this task is based on a convention, it could be automated. Unfortunately, the current integration between the CM and the bug-tracking tool does not support that. That is a major source of complaints repeatedly reported by the MVP software developers during the interviews.

6.3. Support for E-mail Conventions

NASA requires ISO 9001 certification for all software development efforts, which means that all changes in the software must be documented, reviewed, and formally authorized before the changes are integrated in the code. In other words, developers need to be accountable for their work. The MVP team chose to use e-mail as a formal communication channel in the organization, as clearly mentioned in the software development process. Indeed, some of the tasks (such as requesting and answering code reviews) were performed by using e-mail. These tasks require the use of software development tools such as source-code editors, CM tools, and so on. Unfortunately, e-mail is not integrated with these tools, which means that developers need to move back and forth between e-mail and the other tools in order to get their work done. Integration of e-mail with software development technology seems easy to implement; it is also very promising because more and more software development organizations are seeking certifications such as ISO 9001 and CMM (Capability Maturity Model). This aspect was identified during the field work and later corroborated by MVP software developers during the interviews. In addition, e-mail messages exchanged among developers are also used to identify expertise in parts of the source code, as well as a history mechanism to identify changes that happened in the past. Again, this information could and should be properly organized and indexed in order to facilitate these activities.

6.4. Holding onto Check-ins

The informal approach of holding onto check-ins is used to avoid disrupting others' work. The support for this task provided by CM tools is appropriate because these tools allow a developer to check files in or out and merge different versions of them at any time. However, this approach is useful only if the developer who is going to check-in some code is aware that his or her work will cause the recompilation of other files. This suggests that software visualization tools (e.g., [4]) that use existing information from the CM tool could be used to support the identification of these files by novice developers who are not aware of the interdependencies in the source code.

6.5. Partial Check-ins

A check-in is called "partial" by the MVP developers when it is performed without a code review to avoid several "back merges" due to the file being changed by several other developers at the same time. CM tools support partial check-ins because they usually do not impose constraints about when check-ins might be performed, allowing one to check-in code into the repository at any time. However, the current trend of integrating CM tools with software process technology [5] might disrupt that. We recognize this integration is essential because it allows the efficient automation of repetitive tasks (such as building a software release) [12]. Nevertheless, the enforcement of the process that usually goes along with this integration must be managed, because it has long been recognized as problematic [29]. CM tools must be flexible enough to allow software developers to use workarounds that deviate from the process in order to properly deal with the problems that they face. One example of such workarounds is the partial check-in. Another approach is to update the software development process to reflect the need for partial check-ins, and consequently legitimate them. In this case, similar to holding check-ins, the information already present in the CM tool could be used by software visualization tools [4] to allow novice developers to identify files with a high degree of parallel development that need to be partially checked-in.

6.6. Speeding Up the Process

MVP developers rush their activities during the development process to minimize the number of dependencies between their code and recently committed changes in the repository (section 5.5). Current CM and bug-tracking tools create the need to speed up because they shield a developer's workspace from other developers' workspaces to support parallel development. Although it is desirable to isolate one developer's work from others, it does not allow developers to coordinate their check-ins, and hence avoid the need to re-do their work. To the best of our knowledge, no existing software engineering tool solves this problem. However, a promising approach recently emerged with tools that attempt to break the isolation of CM workspaces (e.g., [24] and [17]). These tools achieve that by distributing the CM commands happening in a developer's workspace to other selected workspaces. These tools focus on the actions of the developers (conveyed as CM commands) because they want to avoid conflicts between the files that two or more developers have checked-out. In addition, we argue that these tools need to provide information about the "status" of other developers' work. By doing that, they allow a developer to identify who is about to check-in code into the repository and, therefore, to coordinate their work, so that a developer does not need to rush. We believe that this can be achieved by extending these tools to collect information from sources other than the CM tool, such as e-mail, the bug-tracking tool, the software process specification, and so on.

7. Discussion

As mentioned before, a formal process description can never completely represent all variations that might occur in a software development effort [8]. Therefore, as the data have suggested, informal approaches need to be adopted to complement the formal approaches to properly support the management of the interdependencies that occur in the software development process. However, to properly support cooperative software development, we need to unveil these informal approaches and provide computational support for them to minimize errors and improve their performance. One of the reasons these informal approaches are important is the high level of parallel development that occurs in large-scale collaborative efforts [20]. Indeed, the engagement in parallel development identified in this field study helps to substantiate the results of Perry et al. [20] that describe high levels of parallel development, but contrasts with the groups studied by Grinter [9, 11], in which developers avoided this situation. Technical improvements in merging techniques from 1995 to 2002 [2] might be the cause of divergence from Grinter's earlier observations. Grinter, however, does not clearly describe the branching strategy used by the team studied, whereas the MVP team adopted the "branch-by-purpose" strategy. According to Walrad and Strom [31] this "strategy supports a high level of parallel development by allowing developers to work on different branches at the same time. Therefore, this might be another explanation for the difference between the two groups. Finally, an organization's structural properties (e.g., reward systems, policies, norms, and so on) are other factors that influence the adoption and use of col-

laborative tools [18]. The two organizations studied are different, hence they are very likely to have different structural properties, which might explain the different levels of engagement in parallel development.

Meanwhile, this field study supports Grinter's [9] finding that during parallel development developers will rush to finish their changes. However, while the developers studied by Grinter will speed up because they want to avoid the complexity of merging, MVP developers rush because they do not know when another developer might check-in some code that will lead them to another set of tests. In both studies, developers describe their dilemma: they want to produce high-quality code, but they also want to finish their changes fast.

The MVP team needs to perform extra work to successfully manage the interdependencies in the software. This extra work is a form of articulation work necessary to coordinate, negotiate, mesh, and schedule their activities [25]. It is different from recomposition work [10], which is the coordination required to assemble software development artifacts from their parts, because recomposition work focuses on choosing the right components to create a software artifact due to source-code dependencies, whereas this extra work focuses on the management of all dependencies that exist in a software development effort.

Finally, in this informal field study we identified another approach used by software developers to identify experts. Whereas McDonald and Ackerman [16] describe the usage of change history data (equivalent to PRs in the MVP team), novice developers in the MVP team use the broadcasted e-mail messages prescribed by the software development process. The importance of finding experts for problem-solving in any organization and the complexity of the MVP code suggest that the operation of sending e-mail before a check-in is essential.

8. Conclusion and Final Remarks

This paper reports the findings of an informal field study conducted at the NASA/Ames Research Center during the course of an eight-week internship with a software development. The results of this field study describe the formal and informal practices adopted by team members to manage the interdependencies that occur during software development. Formal approaches are those legitimated by the organization; the informal ones are those that emerge naturally due to the needs of the developers. Examples of formal approaches adopted by the team are the software development process, some software development tools, design meetings, and a clear division of labor. The informal approaches that we identified are partial check-ins, problem reports that cross work boundaries, holding onto check-ins, e-mail and naming conventions, and the action of speeding up the processes.

In this work, we also indicate current and nonexisting computational support to the informal approaches. Indeed, partial check-ins, problem reports that cross work boundaries, and holding onto check-ins are work practices currently supported by CM and bug-tracking tools. E-mail and naming conventions and the action of speeding up the processes are adopted by MVP developers due to the lack of tool support. We believe that these interesting research areas should be further investigated. Pointing out these areas is an important contribution of this paper.

Finally, we are planning a future study in a different organization. We seek to identify similarities and differences in the formal and informal approaches that we identified here and to learn how the ones that we identified are used in a different context.

Acknowledgments

The authors thank CAPES (grant BEX 1312/99-5) and NASA/Ames for financial support. This effort was also sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Funding also was provided by the National Science Foundation under grant numbers 0205724 and 0083099. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force Laboratory, or the U.S. Government.

9. References

[1] Appleton, B., Berczuk, S., et al., "Streamed Lines: Branching Patterns for Parallel Software Development," Proceedings of Pattern Languages of Programs (PLoP'98), Washington University Technical Report #WUCS-98-25, 1998.

[2] Conradi, R., and Westfechtel, B., "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, pp. 232-282, 1998.

[3] Curtis, B., Krasner, H., et al., "A Field study of the Software Design Process for Large Systems," *Communications of the ACM*, vol. 31, pp. 1268-1287, 1988.

[4] Eick, S. G., Graves, T. L., et al., "Visualizing Software Changes," *Software Engineering*, vol. 28, pp. 396-412, 2002.

[5] Estublier, J., "Software Configuration Management: A Roadmap," Future of Software Engineering, pp. 279-289, Limerick, Ireland, 2001.

[6] Finkelstein, A., Kramer, J., et al., *Software Process Modeling and Technology*: Wiley, 1994.

[7] Gamma, E., Helm, R., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addi-

son-Wesley, 1995.
[8] Gerson, E. M., and Star, S. L., "Analyzing Due Process in the Workplace," *ACM Transactions on Office Information Systems*, vol. 4, pp. 257-270, 1986.
[9] Grinter, R., "Supporting Articulation Work Using Configuration Management Systems," *Computer Supported Cooperative Work*, vol. 5, pp. 447-465, 1996.
[10] Grinter, R. E., "Recomposition: Putting It All Back Together Again," Conference on Computer Supported Cooperative Work (CSCW'98), pp. 393-402, 1998.
[11] Grinter, R. E., "Using a Configuration Management Tool to Coordinate Software Development," Conference on Organizational Computing Systems, pp. 168-177, 1995.
[12] Grinter, R. E., "Workflow Systems: Occasions for Success and Failure," *Computer Supported Cooperative Work*, vol. 9, pp. 189-214, 2000.
[13] Kraut, R. E., and Streeter, L. A., "Coordination in Software Development," *Communications of the ACM*, vol. 38, pp. 69-81, 1995.
[14] Mark, G., Fuchs, L., et al., "Supporting Groupware Conventions through Contextual Awareness," European Conference on Computer-Supported Cooperative Work (ECSCW '97), pp. 253-268, Lancaster, England, 1997.
[15] McCracken, G., *The Long Interview*: Thousand Oaks, CA: SAGE Publications, 1988.
[16] McDonald, D., and Ackerman, M., "Just Talk to Me: A Field Study of Expertise Location," Conference on Computer Supported Cooperative Work, pp. 315-324, 1998.
[17] O'Reilly, C., Morrow, P., et al., "Improving Conflict Detection in Optimistic Concurrency Control Models," 11th International Workshop on Software Configuration Management (SCM-11), Portland, Oregon, 2003.
[18] Orlikowski, W., "Learning from Notes: Organizational Issues in Groupware Implementation," *The Information Society*, vol. 9, 1993.
[19] Parnas, D. L., "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, pp. 1053-1058, 1972.
[20] Perry, D. E., and, Siy, H. P., et al., "Parallel Changes in Large-Scale Software Development: An Observational Case Study," *ACM Transactions on Software Engineering and Methodology*, vol. 10, pp. 308-337, 2001.
[21] Perry, D. E., Staudenmayer, N. A., et al., "People, Organizations, and Process Improvement," *IEEE Software*, vol. 11, pp. 36-45, 1994.
[22] Podgurski, A., and Clarke, L. A., "The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance," Symposium on Software Testing, Analysis, and Verification, pp. 168-178, 1989.
[23] Rothermel, G. and Harrold, M. J., "A Safe, Efficient Regression Testing Selection Technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 173-210, 1997.
[24] Sarma, A., Noroozi, Z., et al., "Palantír: Raising Awareness among Configuration Management Workspaces," Twenty-fifth International Conference on Software Engineering, pp. 444-453, Portland, Oregon, 2003.
[25] Schmidt, K., and Bannon, L., "Taking CSCW Seriously: Supporting Articulation Work," *Journal of Computer Supported Cooperative Work*, vol. 1, pp. 7-40, 1992.
[26] Shull, F., Carver, J., et al., "An Empirical Methodology for Introducing Software Processes," Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 288-296, Vienna, Austria, 2001.
[27] Star, S. L., and Griesemer, J. R., "Institutional Ecology, Translations and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology," *Social Studies of Science*, vol. 19, pp. 387-420, 1989.
[28] Strauss, A., and Corbin, J., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, Thousand Oaks, CA: SAGE publications, 1998.
[29] Suchman, L., *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge: Cambridge University Press, 1987.
[30] Vessey, I., and Sravanapudi, A. P., "CASE Tools as Collaborative Support Technologies," *Communications of the ACM*, vol. 38, pp. 83-95, 1995.
[31] Walrad, C., and Strom, D., "The Importance of Branching Models in SCM," *IEEE Computer*, vol. 35, pp. 31-38, 2002.

Opportunities for Extending Activity Theory for Studying Collaborative Software Development

Cleidson R. B. de Souza[†] and David F. Redmiles
School of Information and Computer Science
University of California, Irvine
{cdesouza, redmiles}@ics.uci.edu

Abstract

Activity theory is an analytical framework that has been used successfully to understand and explain collective work. Software development is, of course, one particular kind of collective work. We used activity theory to analyze the observations one author made during an internship with a large-scale software development group. We also made some observations about how well suited activity theory was for the analysis. We briefly describe the work setting and the analysis. Then we describe the experiences we had, which indicate possibilities for further developing activity theory for studying collaborative work.

1. An Experience with Collaborative Software Development

The first author spent eight weeks during the summer of 2002 interning as a software developer on a large-scale software development team. As a member of this team, he was able to make observations and collect information about a variety of aspects, including the organization of the team, the formal and informal practices that this team adopted, and the tools they used. The software development team was formed to develop a software application we call MVP (not the real name), which comprises 10 different tools that are deployed in different parts of the United States. The source code is approximately one million lines of C and C++.

Each of the several different tools that compose MVP uses a specific set of "processes." A process for the MVP team is a program that runs with the appropriate run-time options. Processes typically run on distributed Sun workstations and communicate by using a TCP/IP socket protocol. Running a tool means running the processes required by this tool, with their appropriate run-time options.

The software development team is divided into two groups: the developers and the verification and validation (V&V) staff. The developers are responsible for writing new code, fixing bugs, and adding new features. This group comprises 25 members. The V&V staff are responsible for testing and reporting bugs identified in the software, keeping a running version of the software for demonstration purposes, and maintaining the documentation (mainly user manuals) of the software. This group comprises six members.

The MVP group adopts a formal software development process that prescribes the steps that need to be performed by the MVP developers during the software development activities. For example, all developers, after finishing the implementation of a change, should integrate their code with the main baseline. In addition, each developer is responsible for testing the code to verify that his/her integration did not insert bugs in the code, or "break the code," as this is informally characterized by MVP developers. After using a configuration management (CM) tool to check-in files into the repository, a developer must send an e-mail to the software development mailing list describing the problem report (PR) associated with the changes, the files that were changed, and the branch where the check-in will be performed, among other pieces of information.

2. An Activity Theory Analysis

Activity theory allows a variety of ways to analyze phenomena. In this work, Engeström's activity theory model [4] was used in the analysis of findings. This model is presented in Figure 1. Activities are associated with objectives called "outcomes." People working within a community share activities. They work to create objects and rely on tools referred to as artifacts to support their activity. Rules instantiate division of labor and practices of the community.

Figure 2 is basically an "instantiation" of the framework described in Figure 1 as applied to the MVP software development team. The main outcome of the software development activity is high-quality MVP software (i.e., bug-free software that is easy to evolve, delivered on schedule, and meets the customers' specifications). Of course, this includes executables, source code, bug repositories, manuals, specifications, and so on. The *object* of this activity is the MVP software while being modified. This includes, for example, the changes being introduced in the code, reported bugs not yet solved, and so on. The *mediating artifacts*, or *tools*, are the set of tools used by the team to manipulate the object so they achieve their goal or outcome, such as CM and bug tracking tools, e-mail, and the like. *Rules* consist of formal practices

[†] Also at the Department of Informatics, Universidade Federal do Pará, Belém, PA, Brazil.

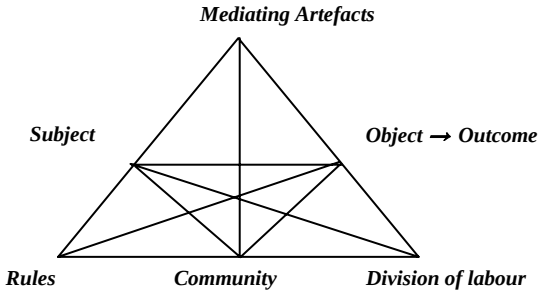


Figure 1: Elements of the Activity Theory Framework (see [4]**).**

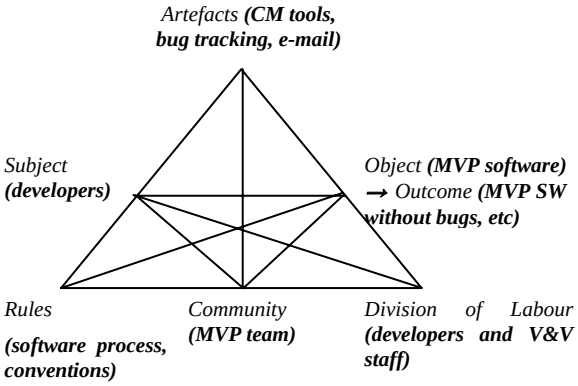


Figure 2: The Software Development Activity as Applied to the MVP Team

(e.g., software development processes) and informal practices (conventions, workarounds, and so on) used by the MVP team. The *community* is the whole MVP team, which is organized according to a specific *division of labor*: There are mainly two groups, namely, developers and V&V staff. But the members of these groups also adopt a division of labor. Specifically, there are process leaders and process developers, the configuration and release manager, the software manager, and testers.

2.1. Tensions and Their "Fixes" in the MVP Team

Contradictions are important aspects in an activity because they might be used as sources of development ([6], pg. 34). In other words, contradictions trigger reflection, thereby helping in the improvement of the activity. Contradictions reveal themselves as breakdowns, problems, tensions, or misfits between elements of an activity or between activities. In our case, we identified several *tensions* within the software development activity developed by the MVP team, but, in addition to that, we also identified the *fixes* that the team adopted to solve them. We identified tensions between different elements, such as between the object and the community, and between the rules and the community.

In the first case, the tension between the object and the community exists due to the effects that the object (e.g., changes in the MVP software) will have on the community. For example, if a change (the object) is introduced in the source code, other members of the MVP team (the community) might need to be informed because they may need to perform additional tasks (e.g., update the documentation) due to that change. The tension exists because developers are not aware of some interdependencies in the software and, therefore, how other members of the community are affected by their work. Despite that, the community must support the evolution of the software and guarantee that the software delivered is not inconsistent with the specifications, manuals and other artifacts.

In the second case, the tension exists basically between rules and the community because one rule suggests that a developer should perform a specific action, but he/she does not want to perform that action out of concern for the effects of the action on the rest of the community. For example, if one developer decides to check-in his/her code into the repository, the other developers (part of the community) might need to recompile their code in order to work with the latest version of the software, and this compilation process is time-consuming.

2.2. Tensions between the Object and the Community

In this case, tensions emerge in the software development activity due to the concern about how the object will affect the community. For example, when the source code is modified, often it is also necessary to modify other software artifacts, such as manuals, documentation, specifications, and so on, or inconsistencies will arise. Although inconsistencies might have positive effects in software development, in general they are not desirable [10]. The MVP software development team already recognized the need to handle this problem (tension) and adopted two different and complementary practices to deal with it: Formal reviews are adopted in the software development process to handle inconsistencies in the source code, and problem reports are structured in such a way that the inconsistencies between source code and other artifacts are easier to manage. Both practices are explained in the following sections.

2.3. Tensions between the Rules and the Community

These tensions occur because a rule might suggest that a developer should perform a specific action, but the developer does not want to perform it due to concern about the effect of this action on the community. As mentioned earlier, an example of such tension occurs when one developer needs to check-in his/her code into the repository, but the other developers would then need to recompile their code in order to work with the latest

their code in order to work with the latest version of the software. Because this compilation process is time-consuming, the developer needs to decide whether to follow the rule and thus cause the whole community to recompile, or to not follow the rule, at least for a while, thereby minimizing the impact of his/her actions in the rest of the community. Typical fixes adopted by the MVP team include changing the order in which some rules are executed or performing additional actions along with the rule to minimize the disruption to the community.

Furthermore, tensions between these components also arise due to the impact on the community in the execution of the rule. In other words, the developer is concerned that he/she needs to perform a rule but actions of the community (such as check-ins or check-outs) will impact his/her performance of the rule. In this case, those actions influence how the developer performs the rule. Note that in this case, the division of labor also influences this tension because it prescribes how developers should be organized in the community, therefore allowing two or more developers to work and check-in in concurrently.

3. Implications for Activity Theory

3.1. Modeling Human Activity

In software development terms, section 2 of this paper developed a model. The process of developing this model has more similarities to software modeling than one might expect. In particular, we began by choosing a modeling language that seemed appropriate for our application—the language of activity theory, and in particular Engeström's terminology and diagrammatic notation. We then built an instance of a model in this language that served as a first approximation. We then refined it through several iterations. We reached a point at which analysis of the model yielded explanations consistent with the data, as presented above.

Iterative refinement of the model appeared to be an open-ended process. However, the actual observations made during the internship acted in a sense like a "test oracle." Namely, we reached a stopping point when all observed phenomena were accounted for. Moreover, the focus of activity theory on identifying tensions and conflict were useful for understanding what we observed and for highlighting areas where software tools and practices might be improved.

In sum, the attempt to model the human collective activity of collaborative software development did not seem straightforward at first, but required a first approximation and successive refinement. Although frustrating, the challenges did not seem greater than other kinds of modeling, and the results were informative. In the next subsection, we make some observations on how this process may be improved and identify research areas for the methodology.

3.2. Activity Theory: Where Next?

Activity theory has been applied to the design of software systems, and research to date has indicated its usefulness toward collecting requirements for software system design (e.g., [1] and [8]). However, to the authors' knowledge, this paper represents the first application of activity theory to studying collaboration among software developers; previous studies have examined only the collaboration between end users and software developers. Thus, we had to struggle with a finer degree of detail of activity than previous works with respect to the development of software.

One challenge that presented itself was the notion that a single activity might be consistent when observed as a single instance, but might be a source of tension when there were multiple instances of that activity. For instance, in the case of a single developer, even when working with end users and other team members, the activity of checking-in a module revision is consistent within itself. However, multiple instances of this check-in activity create a tension we observed as developers sped up their work to be the first to check-in. This part of the model and the more general issue of multiple instances of activity is one place for further research into the application of activity theory and a potential contribution to improving the methodology.

Another area for research in activity theory is akin to dependency analysis in software testing. Namely, as we identified different activities that comprised the general activity of evolving a software system, we began to observe many interdependencies. For example, rules for applying a specific software tool led to other activities, each with their own associated set of rules, subjects, other tools, and so on. We were intrigued by the notion that a kind of dependency analysis might be developed to help an organization more precisely account for the potential impact of making changes to tools and practices. This kind of work, however, would be a long-term goal. A related issue is that of adoption. Understanding the history of how elements in the activity theory models evolved (e.g., tools, rules, division of labor, and so on) can better enable the responsible introduction of new tools, including involving end users with tool introduction. The basic premise of introducing changes into people's work is the ability to develop the fullest understanding possible of that work. Activity theory, even in its present state of development, is successful in that regard.

Finally, a new line of research is beginning to present itself around the concepts of reflection and awareness. Specifically, various researchers have begun to recognize the value of simply reflecting back to a group or organization the actuality of its various objectives and activities. In a previous study, we used this kind of reflection as a matter of course in reporting findings, but the process of performing this "reporting" led to improvement in the

process of software developers collecting requirements and in the organization's members better understanding one another's roles [2]. Other researchers have observed similar effects, including those at a small scale. Namely, some researchers are developing software tools to help people coordinate their collaborative work by reflecting the current state of a collaborative activity or the state of actual collaborators. Some instances are Portholes systems that reflect the state of collaborators [3] [7], configuration management tools that reflect who is working on what modules [9], and tickertape tools that reflect all activities in a work environment [5]. Thus, another open area is better understanding and better reflecting of actual activity (through manual and automated means) back to participants in that activity, and understanding ways this has positive effects on the collective work.

4. Conclusions

Our experiences in performing the analysis presented briefly in this paper as well as previous experiences of our own and our colleagues have shown many positives to activity theory. It is open ended, which, although a challenge, allows for the introduction of new ideas and refinements. It is noninvasive, using open-ended interviews or even more informal observations of work such as presented in this paper. It readily yields to iterative refinement. When more detail is needed in a model, additional activities may be named and analyzed. Finally, there seems to be some overlap in object-oriented analysis. Although the present authors do not wish to overemphasize the similarities, the overlap is helpful for people with object-oriented experience to engage in learning the methodology. Thus, although there is still a great deal of craft involved in becoming acquainted with and applying activity theory, we have experienced many positives in our analyses in different work settings and anticipate the methodology becoming more refined and documented.

Acknowledgments

The authors thank CAPES (grant BEX 1312/99-5) and NASA/Ames for financial support. This effort was also sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Funding also was provided by the National Science Foundation under grant numbers CCR-0205724 and 9624846. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

5. References

[1] Bodker, S., *Through the Interface: A Human Activity Approach to User Interface Design*, Hillsdale, NJ: Lawrence Erlbaum, 1991.

[2] Collins, P., Shukla, S., et al., "Activity Theory and System Design: A View from the Trenches," *Computer Supported Cooperative Work—Special Issue on Activity Theory and the Practice of Design*, vol. 11, pp. 55-80, 2002.

[3] Dourish, P., and Bly, S., "Portholes: Supporting Distributed Awareness in a Collaborative Work Group," ACM Conference on Human Factors in Computing Systems (CHI '92), Monterey, CA, 1992.

[4] Engeström, Y., "Activity Theory and Individual and Social Transformation," pp. 19-38, in Engeström, Y., Miettinen, R., and Punamäki, R-L., "Perspectives on Activity Theory." Cambridge, UK: Cambridge University Press, 1999.

[5] Fitzpatrick, G., Mansfield, T., et al., "Augmenting the Workaday World with Elvin," 6th European Conference on Computer Supported Cooperative Work, pp. 431-450, Copenhagen, Denmark, 1999.

[6] Kuuti, K., "Activity Theory as a Potential Framework for Human-Computer Interaction Research," pp. 17-44, in Nardi, B., "Context and Consciousness: Activity Theory and Human-Computer Interaction." Cambridge, MA: The MIT Press, 1996.

[7] Lee, A., and Girgensohn, A., "NYNEX Portholes: Initial User Reactions and Redesign Implications," ACM Conference on Human Factors in Computing Systems (CHI '97), pp. 385-394, 1997.

[8] Nardi, B., and Redmiles, D., Eds. Computer Supported Cooperative Work, *The Journal of Collaborative Computing, Special Issue on Activity Theory and the Practice of Design*, Vol. 11, No. 1-2, p. 1-11, 2002.

[9] Sarma, A., Noroozi, Z., et al., "Palantír: Raising Awareness among Configuration Management Workspaces," Twenty-fifth International Conference on Software Engineering, pp. 444-453, Portland, Oregon, 2003.

[10] Spanoudakis, G., and Zisman, A., "Inconsistency Management in Software Engineering: Survey and Open Research Issues," in *Handbook of Software Engineering and Knowledge Engineering*, vol. 1, S. K. Chang, Ed.: World Science Publishing Co., 2001, pp. 329-380.

“Breaking the Code”, Private and Public Work in Collaborative Software Development

Cleidson R. B. de Souza[1,2] and David F. Redmiles[2]
[1]Universidade Federal do Pará, Brazil and [2]University of California, Irvine, USA
cdesouza@ics.uci.edu, redmiles@ics.uci.edu

Abstract. As a cooperative effort, software development is especially difficult because of the many interdependencies amongst the artifacts created during this activity. In order to minimize problems created by these interdependencies, some software development tools create a distinction between private and public aspects of work of the developer. Technical support is provided to these aspects as well as for transitions between them. However, we present empirical material collected from a software development team that suggests that the transition from private to public work needs to be more carefully handled. Indeed, our analysis suggests that different formal and informal work practices are adopted by the developers to allow a delicate transition, where software developers are not largely affected by the emergent public work.

Private and Public Work in CSCW

Software engineers have sought for quite some time to understand their own work of software development as an important instance of cooperative work, especially seeking ways to provide better software tools to support developers (Curtis, Krasner et al. 1988). Indeed, they created different tools, such as configuration management (CM) and bug tracking systems, to facilitate the coordination of groups of developers (Grinter 1995). However, software development is especially difficult as a cooperative endeavor because of the several interdependencies that arise in any software development effort. To minimize these problems, CM systems adopt design constructs (like branches and workspaces used in configuration management systems) to shield each individual from effects of other developers’ work. These workspaces enforce a distinction between the *private* aspects of work developed by the software engineer and the *public* aspects that occurs when this developer shares his work with the other developers. Similar approaches have been taken in other categories of collaborative applications (e.g., collaborative writing and hypermedia systems), which have adopted this distinction between private and public work in order to facilitate collaboration. This is usually done through the provision of separate private and public (or shared) workspaces. Private workspaces allow users to work in different parts of a document in parallel and contain information that only one user can see and edit allowing him to create drafts that later will be shared with the other co-workers. On the other hand, public workspaces allow all users to share the same information or document.

When support for private and public work is provided, it is also necessary to support *transitions* between them. The central issue in systems maintaining separate workspaces is how information or

activity moves between them, and similarly, the central mechanism around which CM systems are built is the mechanism for moving information between public and private conditions – checking in, checking out, merging. In cooperative working settings, people selectively choose *when* and *how* to disclosure their private work to others, i.e., they want to be able to control the emergence of public information (Ackerman 2000). CM tools and collaborative authoring tools provide support for these transitions. In collaborative writing, for example, one can basically copy the content of a private workspace and paste into the public workspace. On the other hand, in CM systems, more sophisticated tools involving merging algorithms and concurrency control policies need to be used because of the aforementioned interdependencies in the software.

Transitions between private and public work (and vice-versa) are particularly important in cooperative work and can lead to problematic situations when overlooked. Indeed, Sellen and Harper (Sellen and Harper 2002) describe some case studies of companies that had problems because they underestimated the delicacy of this transition. Despite that, insufficient analytical attention has been given to this transition by the CSCW community. In this paper, we will examine this issue with empirical material collected from a collaborative software development effort. The team observed used three software development tools for coordination purposes. However, these tools alone were not sufficient to effectively support the team; participants needed to adopt a set of formal and informal work practices to properly support private, public work and transitions between them. The adoption of these different work practices suggests that the computational support provided by these systems to support the emergence of private information is still unsatisfactory.

Setting and Methods

The group studied develops an application called MVP (not the real name) and is divided in two teams: developers and the verification and validation staff (V&V). Developers are responsible for writing new code, for performing bug fixing, enhancements, and so on. There are 25 developers, including researchers that write their own code. The V&V team (6 engineers) is responsible for testing the software, keeping a running version for demonstration and maintaining user manuals.

The first author spent eight weeks during the summer of 2002 as a member of the MVP team. During that time, he was able to interview developers, make observations and collect information about several aspects of the team. He also talked with his colleagues to learn more about their work. Additional material was collected by reading manuals of the MVP tools, manuals of the software development tools used, formal documents (like the description of the software development process and the ISO 9001 procedures), problem reports (PR's), and so on.

MVP Practices to Handle Private and Public Work

The main tools used by the MVP team to coordinate their activities are the configuration management (CM) and the bug tracking tools (Grinter 1995). Branching in CM tools are used to create shields between developers' workspaces isolating one's work from others (Conradi and Westfechtel 1998). On the other hand, merging mechanisms are created to allow one's work to be combined with other developers' work. In other words, branches support private work, while merging mechanisms support the transition from private to public work. Finally, building mechanisms in CM tools support the public work because they allow developers to automatically recompile the code in order to incorporate changes recently committed in the repository.

In general, we identified that the private and public work are properly supported by the software development tools and by the software development process adopted by the MVP. However, except for

the merging mechanisms embedded in CM tools, the *transitions* between private and public are improved through informal work practices because of the need developers have to manage the interdependencies. Examples of these practices will be briefly discussed in the following paragraphs.

We called the first practice “holding onto check-in’s”. Developers will hold onto check-in’s (and merges) when they realize that their work (in this case, their changes in the software) will imply in the recompilation of the whole source code. They avoid that because they know that the recompilation process is time-consuming usually taking between 30 to 45 minutes. This means that other developers will waste their time waiting for the recompilation of their local copies.

After making their work public by merging it back into the repository, the software development process prescribes that MVP developers must send an e-mail to the whole software development group informing about the new changes in the system. However, these developers will send this e-mail before committing their changes and will also add a brief description of the impact that these changes will cause on their colleague’s work. In this case, because of these e-mail messages, other developers might reflect about the effect of their colleagues’ changes in their current work and prepare for that. This is possible because they are aware of some interdependencies in the source-code. The convention (adding impact statements in the e-mails) is the second practice identified.

A third approach identified was the “partial check-in”, which consists of checking-in files back in the repository, even when the developers have not yet finished their entire work. This is used to deal with parallel development in files that are changed very often. This practice allows developers to reduce the work necessary to make their work public, as it minimizes the number of updates that they need to perform in their files before merging them into the main repository.

Finally, we also identified that problem reports (PR’s) are used by different stakeholders (e.g., end-users liaisons, developers and testers) to manage the software interdependencies. For example, when a bug is identified, it is associated with a specific PR. Whoever identified the problem is also responsible for filling in the PR with information about ‘how to repeat’ it. This description is used by the developer assigned to fix the bug to specify the circumstances (adaptation data, tools and their parameters) under which the bug appears. In short, MVP members use the information from the PR’s in many different ways, according to the role they are playing.

Conclusions

We briefly described some of the work practices adopted by software developers to properly handle the transition between their private and their public work. MVP developers employ these practices because of the interdependencies that exist in the software. As mentioned before, the adoption of these practices suggests that computational support is necessary in cooperative software development tools to support the emergence of private information.

References

Ackerman, M. S. (2000). "The Intellectual Challenge of CSCW: The Gap Between Social Requirements and Technical Feasibility." Human-Computer Interaction **15**(2-3): 179-204.

Conradi, R. and Westfechtel, B. (1998). "Version Models for Software Configuration Management." ACM Computing Surveys **30**(2): 232-282.

Curtis, B., Krasner, H. and Iscoe, N. (1988). "A field study of the software design process for large systems." Communications of the ACM **31**(11): 1268-1287.

Grinter, R. E. (1995). Using a Configuration Management Tool to Coordinate Software Development. Conference on Organizational Computing Systems, Milpitas, CA, 168-177.

Sellen, A. J. and Harper, R. H. R. (2002). The Myth of the Paperless Office. Cambridge, Massachusetts, The Mit Press.

"Breaking the Code", Moving between Private and Public Work in Collaborative Software Development

Cleidson R. B. de Souza[1,2] David Redmiles[1] Paul Dourish[1]

[1]School of Information and Computer Science
University of California, Irvine
Irvine, CA, USA – 92667

[2]Departamento de Informática
Universidade Federal do Pará
Belém, PA, Brazil - 66075

{cdesouza,redmiles,jpd}@ics.uci.edu

ABSTRACT

Software development is typically cooperative endeavor where a group of engineers need to work together to achieve a common, coordinated result. As a cooperative effort, it is especially difficult because of the many interdependencies amongst the artifacts created during the process. This has lead software engineers to create tools, such as configuration management tools, that isolate developers from the effects of each other's work. In so doing, these tools create a distinction between private and public aspects of work of the developer. Technical support is provided to these aspects as well as for transitions between them. However, we present empirical material collected from a software development team that suggests that the transition from private to public work needs to be more carefully handled. Indeed, the analysis of our material suggests that different formal and informal work practices are adopted by the developers to allow a delicate transition, where software developers are not largely affected by the emergent public work. Finally, we discuss how groupware tools might support this transition.

Categories and Subject Descriptors

H.4.1 [**Office Automation**]: Groupware; H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work;

General Terms

Human Factors

Keywords

Private work, public work, collaborative software development, qualitative studies.

1. INTRODUCTION

Software engineers have sought for quite some time to understand their own work of software development as an important instance of cooperative work, especially seeking ways to provide better software tools to support developers [6]. Indeed, they created several different tools, such as configuration management (CM) and bug tracking systems, to facilitate the coordination of groups of developers [14]. However, software development is especially difficult as a cooperative endeavor because of the several interdependencies that arise in any software development effort. To minimize these problems, current CM systems adopt design constructs (like workspaces and branches used in configuration management systems) to shield each individual from effects of other developers' work [5]. These workspaces enforce a distinction between the *private* aspects of work developed by a software engineer and the *public* aspects that occur when this developer shares his work with other developers. Similar approaches have been taken in other categories of collaborative applications (e.g., collaborative writing and hypermedia systems), which have adopted this distinction between private and public work in order to facilitate collaboration. In these applications, this is usually done through the provision of separate private and public (or shared) workspaces. Private workspaces allow users to work in different parts of a document in parallel and contain information that only one user can see and edit allowing him to create drafts that later will be shared with the other co-workers [7]. On the other hand, public workspaces allow all users to share the same information or document and edit it concurrently.

When support for private and public work is provided, it is also necessary to support *transitions* between them. The central issue in systems maintaining separate workspaces is how information or activity moves between them, and similarly, the central mechanism around which CM systems are built is the mechanism for moving information between public and private conditions – checking in, checking out, merging. In cooperative working settings, people selectively choose *when* and *how* to disclosure their private work to others, i.e., they want to be able to control the emergence of public information [1, 26]. CM tools and collaborative authoring tools provide support for these transitions. In collaborative writing, for example, one can basically copy the content of a private workspace and paste into the public workspace. On the other hand, in CM systems, more sophisticated tools involving merging algorithms and concurrency control policies need to be used because of the aforementioned interdependencies in the software.

Transitions between private and public work (and vice-versa) are particularly important in cooperative work and can lead to problematic situations when overlooked. Indeed, Sellen and

GROUP'03, November 9–12, 2003, Sanibel Island, Florida, USA.

Harper [28] describe case studies of companies that had problems because they underestimated the delicacy of this transition. Despite that, insufficient analytical attention has been given to this transition by the CSCW community. In this paper, we will examine this issue with empirical material collected from a collaborative software development effort. The team observed uses mostly three tools for coordination purposes: a configuration management tool, a bug-tracking system, and e-mail. However, these tools alone were not sufficient to effectively support the team; participants needed to adopt a set of formal and informal work practices to properly support private, public work and transitions between them. The adoption of these different work practices suggests that the computational support provided by these systems to support the emergence of private information is still unsatisfactory. Based on these results, we draw more general conclusions about the implications for computer-supported cooperative work.

The rest of the paper is organized as follow. The next section discusses the idea of private and public work in computer-supported cooperative work. Then, sections 3 and 4 present the settings and the methods that we used to study the software development team. After that, Section 5 describes the set of work practices adopted by the team to properly deal with private, public work and transitions between them. Section 6 presents our discussion about the data that we collected. After that, Section 7 discusses implications of our findings in the design of CSCW tools. Finally, conclusions and ideas for future work are presented.

2. PRIVATE AND PUBLIC WORK

In this paper we examine the distinction between private and public work in collaborative efforts. The need for this distinction is widely recognized in CSCW research. According to Ackerman [1], for example, people "(…) have very nuanced behavior concerning how and with whom they wish to share information (…) people are concerned about whether to release this piece of information to that person at this time (…)". Another reason that makes people care about the release of information about them is that they "(…) are aware that making their work visible may also open them to criticism or management (…)" (*ibid.*). Furthermore, one does not make his *entire* work visible because he wants to appear competent in the eyes of colleagues and managers by making their work more complicated than necessary [26]. Indeed, people are not interested in all information that is provided to them. As Schmidt [26] points out:

"(…) in depending on the activities of others, we are 'not interested' in the enormous contingencies and infinitely faceted practices of colleagues unless they may impact our own work (…) An actor will thus routinely expect not to be exposed to the myriad detailed activities by means of which his or her colleagues deal with the contingencies they are facing in their effort to ensure that their individual contributions are seamlessly articulated with the other contributions."

To summarize, people have several contextualized and different strategies to release their *private* information, and they expect that others will do the same, not overloading them with *public* information that is not 'relevant' to their current context or activity. Note that this private information might be collaboratively constructed [16]. In this case, the information is public for those involved in its "construction", but it is private to the other members of the cooperative effort.

CSCW researchers have already recognized the need to support these findings. Indeed, a typical approach to address that is to provide support for private and public (also called shared) windows, or workspaces, to support the collaboration among users [30]. Private workspaces allow users to work in different parts of a document in parallel and contain information that only one user can see and edit, allowing him to create drafts that later will be shared with the other co-workers [7]. On the other hand, public workspaces allow all users to share the same information or document so that, changes in the document are automatically visible to all users. The usage of these workspaces mimic conventions carried over non-technological work, where no one wants to search or look at anyone's private desk or drawer, and conversely wants no one to search theirs, but accepts that when they occur in public spaces. Indeed, Mark and colleagues [21] report how conventions about the use of private and public workspaces implicitly evolved from conventions formed in face-to-face non-technological work after the introduction of a groupware tool.

Often, other mechanisms are present in collaborative systems to make other actions' visible as well. For example, grey 'clouds' were proposed in the collaborative editor Grove to indicate where other co-writers are editing the text [9]. Furthermore, it is also well-known that, in some settings, making others' work public facilitates the coordination of the activities [16] [17] and enables learning and greater efficiencies [20]. Examples of tools that explore such approaches include Portholes [8] and Babble [10].

The underlying distinction between private and public work also implies that in collaborative efforts transitions between these two aspects occur. However, while notions of "public" and "private" have been incorporated into software system design, insufficient analytical attention has been give to the transitions. Field studies such as those of Bowers [4] or Sellen and Harper [28] demonstrate that overlooking these transitions can be problematic. In Bower's study, the disclosure of private data brought about dilemmas of ownership and responsibility among the employees of the organization studied. In Sellen and Harper's study, when the companies tried to go paperless deploying a new information system, the employees' ability to control when to disclosure information was lost and these employees boycotted the system. This happened because paper, as a *medium* on which work was performed, allowed their owners to avoid sharing information with their co-workers until they felt that the information was "ready".

Note that the setting where the collaborative effort takes place is important. For example, in a control room, all workers are collocated, which allows them to use intonations in their voice and/or body language to make their actions visible to other co-workers [17]. On the other hand, Whittaker and Schwarz [34] report an ethnographic study where a large wallboard (containing the schedule of a software development project) is used by the team, which is spread along different cubicles and offices. The public location of this wallboard allowed developers to access

information about who was doing which tasks at which times, among other things. In other words, in this setting, information about others' current actions was made public by checking and updating the schedule displayed in the wallboard.

In collaborative software engineering, this distinction between private and private work is not only desirable, but necessary and often enforced by tools. This occurs because of the several interdependencies that arise in any software development effort. In other words, each part of the software depends, directly or indirectly, on many other parts. Furthermore, these interdependencies are not strictly defined in the artifacts produced, and often are not even known by the developers. To handle this problem, software engineers created tools, such as configuration management (CM) and bug tracking systems, to facilitate the coordination of groups of developers [14]. Current CM systems adopt design constructs (like workspaces and branches) to shield the work of individuals from effects of other developers' work [5]. Basically, these workspaces "create a barrier that prevents developers from knowing which other developers change which other artifacts" [25]. Therefore, CM workspaces allow software developers to work privately. Furthermore, CM systems provide mechanisms to support the transition from private to public work when developers want to make this transition. To be more specific, when a developer finishes his work in his private workspace, he can publicize his work to other software developers through check-in's, check-out's and merging operations. Despite this support, several problems arise in any software development effort. Indeed, based on empirical data that we collected, we identified a set of formal and informal work practices used by a team of software developers to handle these problems. The setting where the data was collected and the methods used to analyze this data are described in the following section.

3. THE SETTING

The team studied is located at the NASA / Ames Research Center and develops a software application we will call MVP (not the real name), which is composed of ten different tools in approximately one million lines of C and C++. Each one of these tools uses a specific set of "processes." A process for the MVP team is a program that runs with the appropriate run-time options and it is not formally related with the concept of processes in operating systems and/or distributed systems. Processes typically run on distributed Sun workstations and communicate using a TCP/IP socket protocol. Running a tool means running the processes required by this tool, with their appropriate run-time options.

Processes are also used to divide the work, i.e., each developer is assigned to one or more processes and tends to specialize on it. For example, there are process leaders and process developers, who, most of the time, work only with this process. This is an important aspect because it allows these developers to deeply understand the process behavior and familiarize with its structure, therefore helping them in dealing with the complexity of the code. During the development activity, managers tend to assign work according to these processes to facilitate this learning process. However, it is not unusual to find developers working in different processes. This might happen due to different circumstances. For example, before launching a new release all workforce is needed to fix bugs in the code, therefore, developers might be assigned to fix these bugs.

3.1 The Software Development Team

The software development team is divided into two groups: the verification and validation (V&V) staff and the developers. The developers are responsible for writing new code, for bug fixing, and adding new features. This group is composed of 25 members, three of whom are also researchers that write their own code to explore new ideas. The experience of these developers with software development range between 3 months to more than 25 years. Experience within the MVP group ranges anywhere between 2½ months to 9 years. This group is spread out into several offices across two floors in the same building.

V&V members are responsible for testing and reporting bugs identified in the MVP software, keeping a running version of the software for demonstration purposes and for maintaining the documentation (mainly user manuals) of the software. This group is composed of 6 members. Half of this group is located in the V & V Laboratory, while the rest is located in several offices located in the same floor and building as this laboratory. Both, the V&V Lab and developers' offices are located in the same building.

3.2 The Software Development Process

The MVP group adopts a formal software development process that prescribes the steps that need to be performed by the developers during their activities. For example, all developers, after finishing the implementation of a change, should integrate their code with the main baseline. In addition, each developer is responsible for testing its code to guarantee that when he integrates his changes, he will not insert bugs in the software, or, "break the code", as informally characterized by the MVP developers. Another part of the process prescribes that, after checking-in files in the repository, a developer must send e-mail to the software development mailing list describing the problem report associated with the changes, the files that were changed, the branch where the check-in will be performed among other pieces of information.

The MVP software process also prescribes the usage of code reviews before the integration of any change, and design reviews for major changes in the software. Code reviews are performed by the manager of each process. Therefore, if a change involves, e.g. two processes, a developer's code will be reviewed twice: one by each manager of these two processes. On the other hand, design reviews are recommended for changes that involve major reorganizations of the source code. Their need is decided by the software manager usually during the bi-weekly software developers meeting (called pre-design meetings).

3.3 Software Development Tools: CM and Bug tracking

MVP developers employ two software development tools for *coordinating* their work: a configuration management system and a bug tracking system. Of course, other tools are used such as CASE tools, compilers, linkers, debuggers and source-code editors, but the CM and bug-tracking tools are the primary means

of coordination [5] [12] [14]. These tools are integrated so that there is a link between the PR's (in the bug tracking system) and the respective changes in the source-code (in the CM tool). Both tools are provided by one of the leader vendors in the market.

A CM tool supports the management of source-code dependencies through its embedded building mechanisms that indicate which parts of the code need to be recompiled when one file is modified. To be more specific, CM tools support both compile-time dependencies, i.e., dependencies that occur when a sub-system is being compiled; and build-time dependencies that occur when several sub-systems or the entire system is being linked [12]. A bug tracking tool, when associated with the CM tool, supports the tracking of changes performed in the source code during the development effort.

It is important to mention that the MVP team employs several advanced features of the CM tool such as triggers, techniques to reduce compilation time, labeling and branching strategies. Indeed, the branching strategy employed is one of the most important aspects of a CM tool because it affects the work of any group of software developers. This strategy is a way of deciding when and why to branch, which makes the task of coordinating parallel changes easier or more difficult [33]. According to the nomenclature proposed by Walrad and Strom [33], the following branching strategies are used by the MVP team: (1) *branch-by-purpose*, where all bug fixes, enhancements and other changes in the code are implemented on separated branches; (2) *branch-by-project*, where branches are created for some of the development projects; and (3) *branch-by-release*, where the code branches upon a decision to release a new version of the product. The branch-by-purpose strategy is employed by MVP developers in their daily work, while the other strategies are only used by the CM manager. In other words, developers create new branches for each new bug fix or enhancement, while branches for projects and releases are created by the manager only. The *branch-by-purpose* strategy supports a high degree of parallel development but at the cost of more complex and frequent integration work [33]. According to this strategy, each developer is responsible for integrating his changes into the main code. This approach is often called "push integration" [2]. After that, the changes are available to all other developers. Therefore, if one bug is introduced, other developers will notice this problem because their work will be disrupted. Indeed, we observed and collected reports of different instances of this situation. When one developer suspects that there is a problem introduced by recent changes, he will contact the author of the changes asking him or her to check the change, or for more information about it.

4. METHODS

The first author spent eight weeks during the summer of 2002 as a member of the MVP team. As a member of this team, he was able to make observations and collect information about several aspects of the team. He also talked with his colleagues to learn more about their work. Additional material was collected by reading manuals of the MVP tools, manuals of the software development tools used, formal documents (like the description of the software development process and the ISO 9001 procedures), training documentation for new developers, problem reports (PR's), and so on.

All the members of the MVP team agreed with the author's data collection. Furthermore, some of the team members agreed to let the intern shadow them for a few days so that he could learn about their functions and responsibilities better. These team members belonged to different groups and played diverse roles in the MVP team: the documentation expert, some V&V members, leaders, and developers. We sampled among MVP "processes", developers' experience in software development and with MVP tools (and processes) in order to get a broader overview of the work being performed at the site. A subset of MVP group was interviewed according to their availability. We again sampled them according to the dimensions explained above. Interviews lasted between 45 and 120 minutes. To summarize, the data collected consists in a set of notes that resulted from conversations, documents and observations based on shadowing developers. These notes have been analyzed using grounded theory techniques [31].

5. PRIVATE AND PUBLIC WORK IN SOFTWARE DEVELOPMENT

As mentioned before, software development tools like configuration management systems support private, public work, and transitions between them. Despite using a CM system the MVP team faced several problems when dealing with these aspects. In this section, we present the formal and informal approaches adopted by this team in order to properly perform their work, i.e. develop software. In the sections that follow, we will explore these situations separately: private work, the transition from private to public, public work, and the transition from public to private.

5.1 Private Work

Configuration management tools allow developers to work privately through the implementation of workspaces and branches [5]. These workspaces isolate the changes being created by one developer from other parts of the code. In this case, a developer's 'work-in-progress' is not shared with other developers. Furthermore, these workspaces allow a developer to work without being affected by the changes of other developers. Indeed, when new changes are committed in the repository by other developers, the CM tool lets the user decide if he or she wants to grab these changes. In case one wants to incorporate the changes, he may recompile the software using the embedded building mechanisms on these tools. In case a developer does not want to incorporate the changes, one can continue working and, if necessary, recompile the software with the appropriate run-time options that do not grab these new changes. Of course, this is a risky course of action because it might lead the developer to work with an outdated version of the files, which might potentially make his work less ineffective.

Mechanisms embedded in CM tools are able to identify syntactic conflicts between the developer's 'work-in-progress' and the changes committed into the repository, reporting whether or not the 'work-in-progress' is affected by these changes. However, because CM systems rely on syntactic features of the domain such as files, suffixes and lines of code, they can not identify semantic conflicts [11]. This means that except for these conflicts, current configuration management systems provide extensive and

automated support for maintaining the isolation between the work performed by one person from other's work [5].

However, when software developers engage in parallel development, problems arise in the CM tool. Parallel development happens when more than one developer needs to make changes in the same file. This means that the same file is checked-out by different developers and all of them are making changes in the different copies of this file in their respective workspaces. As one might imagine, parallel development might lead to conflicts. They might occur when one developer checks-in his changed version of the file back in the repository, because the versions of the other developers will become outdated. In this case, the changes of these developers might become inappropriate because they are based on a code that is not the latest. To solve this problem, a developer needs to update his version of the file by merging the other developer's changes into his code. The developers term this operation "back merging"; in CM terminology, it is named "synchronization of workspaces" or "import of the changes". Conflicting changes are more likely to occur in files that are accessed by several developers at the same time. Indeed, in the MVP software some files are used to describe programming language structures that are used all over the code. This means that several different developers often change these files. In this case, "back merges" are problematic because CM tools face difficulties when they need to perform several merges at the same time. To overcome this problem not avoiding parallel development, MVP developers adopted a strategy to deal with these files: they perform "partial check-in's", which consist of checking-in some of the files back in the repository, even when the developers have not finished all their changes yet. This strategy reduces the number of "back merges" needed, therefore overcoming the limitations of CM tools. In addition, they minimize the likelihood of conflicting changes.

In addition to "partial check-in's", MVP developers adopt a different practice during their private work: they "speed-up" to finish some of their activities during the development process to avoid merging. This does not happen all the time though, it occurs only when MVP developers are testing their changes. This activity is performed right before the check-in operations. As one developer plainly pointed out: "This is a race!". According to the software development process, this testing is necessary to guarantee that the changes will not introduce bugs into the system. We observed that, this testing is very informal: developers will sit on the V&V laboratory and compare the current version of MVP with the one with changes. MVP developers do not use more formal techniques, such as regression testing techniques, at this moment. These will be used by the V&V staff before creating a new release of the software.

In contrast, the bug tracking tool does not provide support for the private work of software developers. All the operations made in the problem reports managed by this tool are publicly accessible to all other software developers. For example, when a developer is assigned a bug, he needs to fill some information about the bug indicating how he will proceed to fix that bug. MVP developers usually write the information to be added to the bug tracking system outside the tool in a private file only accessible by themselves. Eventually, this information is added to the bug-tracking tool by the developer, which will automatically make it available to all members of the MVP team. Furthermore, the tool does not avoid that two developers work on the same PR, as reported by one of the developers. Developers themselves have to deal with this problematic situation. The MVP group tries to avoid this problem through the software development process, which prescribes that the software manager is the one responsible for assigning PR to developers. Any assignment needs the approval of the manager. Organizational rules however interact with this process. According to these rules, the software manager can not assign work to the contractors working for the MVP group. This assignment has to be done to the manager of the contracting company, who will be responsible for assigning the work to the developers.

5.2 Moving from Private to Public Work

In this section we discuss the work practices used by the MVP team to support the transition from private to public work, as well as how the software development tools used by the MVP team support this transition. This transition might occur in two situations: when a developer asks for code reviews, or informal comments, in his code; or when a developer commits his work (source-code changes) into the CM repository.

In the first case, MVP developers want to grant others access to their code, meaning that the work will be visible to them so that they can comment on it. In this case, MVP developers simply need to change a setting in their CM workspaces. Although their work is now *public*, it is *not shared by the other developers*, meaning that it will not impact other developers work.

In the second case, after a developer commits his work into the CM repository, this work is made *public* and *shared* meaning that it is visible and might impact the work of the other developers. In order to publicize his work, the author of the changes has to perform, at least, four different operations[1]:

1. Check-in the files that he wants to publish in his own branch;
2. Check-out the same set of files from the baseline;
3. Merge his changed files with the checked-out files available in the baseline; and
4. Check-in the new files generated by the merging operation into the baseline.

From the technical point of view, these tasks are not difficult since check-in's, check-out's and merges are typical operations in CM systems and, therefore, supported by nearly every tool in the market. This means that CM systems provide adequate support for these operations. However, this support is problematic when a developer is, or was, engaged in parallel development. As mentioned in the previous section, MVP developers adopt "partial check-in's" to deal *only* with files with high levels of parallel development. Other files are not "partially checked-in". In this case, if a developer is engaged in parallel development and other developers had checked-in the same files in the baseline before him, then he will need to perform "back merges" before merging

[1] These operations might be different in other software development teams since they depend on the branching strategy adopted by the team.

his code into the baseline. "Back merges" are supported by the CM tool through the presentation of version trees of the files being merged, which allows developers to identify the need for this task through the observation of the versions on this tree. After that, the operation is a simple merge. Again, the situation becomes problematic only if several "back merges" need to be performed.

During the transition from private to public, there is nothing that other developers need, or are able to do to facilitate this process. The work of performing the transition needs to be done by the author of the changes that will be publicized. However, because of the several inter-dependencies that exist among the several parts of the software (e.g., source-code, manuals, specifications, design documents, and so on), this does not mean that these developers will not be affected by the transition. Indeed, in order to minimize these effects, the developer who is going to perform the transition follows a set of formal and informal practices to facilitate the management of the interdependencies. These practices need to be adopted because the tool support to the developers affected by the private work being publicized is minimal. These formal and informal practices are described below.

The Software Development Process

As mentioned before, the software development process adopted by the MVP team prescribes the usage of code and design reviews. One of the reasons reported by the MVP developers for using these formal reviews is the possibility of evaluating the impact that the changes under review will have on the rest of the code. The most experienced software developer of the team, for example, reported that design reviews are used to guarantee that changes in the code do not "break the architecture" of the MVP software. By breaking the architecture, she means writing code that violates some of the design decisions embedded in the MVP software. Code reviews, on the other hand, are responsibility of process leaders, who can evaluate the impact that the changes will introduce in their processes before they were committed in the main repository. This helps each and every process leader to coordinate the work of other developers working in the same process.

E-mail Conventions

In addition to formal reviews, the MVP process prescribes that *after* checking-in code in the repository, a developer needs to send an e-mail about the new changes being introduced in the system to the software developers' mailing list (see section 3.2). However, we found out that MVP developers send this e-mail *before* the check-in. Moreover, MVP developers add a brief description of the impact that their work (changes) will have on other's work in this e-mail sent to the software developers' mailing list. By adopting these practices, MVP developers allow their colleagues to prepare for and reflect about the effect of their changes. This is possible because all MVP developers are aware of some of the interdependencies in the source-code, but not all of them. As an example of this 'preparation', developers might send e-mail to the author of the changes asking him to delay their check-in, walk to the co-worker's office to ask about these changes or, if the changes have already been committed, browse the CM and bug tracking systems to understand them. The following list presents some comments sent by MVP developers:

> *"No one should notice."*
> "[description of the change]*: only EDP users will notice any change."*
> *"Will be removing the following* [x] *files. No effect on recompiling."*
> *"Also, if you recompile your views today you will need to start your own* [z] *daemon to run with live data."*
> *"The changes only affect* [y] *mode so you shouldn't notice anything."*
> *"If you are planning on recompiling your view this evening and running a MVP tool with live* [z] *data you will need to run your own* [z] *daemon."*

These e-mails are also important because they tell (or remind) developers that they have been engaged in parallel development. Often, developers do not know that this is happening[2]. The information in the e-mail is usually enough to tell the developer if he needs to incorporate these changes right away in order to continue his work, or if he can wait until he is ready for check-in. In both cases, the developer needs to "merge back" the latest changes into his version of the file.

Sending e-mail before a check-in is also used by other developers to support expertise identification, and as a learning mechanism. Developers associate the author of the e-mails describing the changes with the "process" where the changes are being performed. In other words, MVP developers assume that if one developer constantly and repeatedly performs check-ins in a specific process, it is very likely that he is an expert on that process. Therefore, if another developer needs help with that process he will look for him for help:

> "[talking about a bug in a process that he is not expert] *(…) I don't understand why this behaves the way it does. But, most of these PR's seem to have John's name on it. So you go around to see John. So, by just by reading the headline of who does what, you kind of get the feeling of who's working on what (…).So they* [e-mails] *tend to be helpful in that aspect as well. If you've been around for ten years, you don't care, you already know that* [who works with what], *but if you've been here for two years that stuff can really make difference (…)"*

On the other hand, the fact that developers read e-mails sent by other developers to assess the impact of others' changes in their code contributes to their learning experience within MVP. Note that developers who reported the aspects described in this section had little experience working at MVP: the first with 2 years and the second with 2 ½ months.

Problem Reports

The problem reports (PRs) of the bug-tracking tool are used by different members of the MVP team who play diverse roles in the software development process. Basically, when a bug is

[2] Differently than the developers reported by Grinter [14], before checking-out a file, they do not check the version tree that displays information about other developers working on the same file.

identified, it is associated with a specific PR. The tester who identified the problem is also responsible for filling in the PR the information about 'how to repeat' it. This description is then used by the developer assigned to fix the bug to learn and repeat the circumstances (adaptation data, tools and their parameters) under which the bug appears. In other words, the information provided by the tester is then used by the MVP developer to locate, and eventually fix the bug. After fixing the bug, this developer must fill a field in the PR that describes how the testing should be performed to properly validate the fix. This field is called 'how to test'. This information is used by the test manager, who creates test matrices that will be later used by the testers during the regression testing. The developer who fixes the bug also indicates in another field of the PR if the documentation of the tool needs to be updated. Then, the documentation expert uses this information to find out if the manuals need to be updated based on the changes the PR introduced. Finally, another field in the PR conveys what needs to be checked by the manager when closing it. Therefore, it is a reminder to the software manager of the aspects that need to be validated.

In other words, PR's provide information that is useful for different members of the MVP team according to the roles they are playing. They facilitate the management of interdependencies because they provide information to MVP developers that help them in understanding how their work is going to be impacted by the changes that are going to be checked-in the repository.

Holding check-in's

As mentioned earlier, MVP developers add a brief description of the impact of their changes to the e-mail sent to the developers before checking-in any code. Two types of impact statements are used more often than others: changes in run-time parameters of a process, and the need to recompile parts or the whole source code. The former case is important because other developers might be running the process that will be changed with the check-in. The latter case is used because when a file is modified, it will be recompiled, as well as, the other files that depend on it and this recompilation process is time-consuming, up to 30 to 45 minutes. Developers are aware of the delay that they might cause to others. Therefore, they hold check-in's until the evening to minimize the disturbance that they will cause. According to one of the developers:

> *"(…) people also know that if they are going to check-in a file, they will do in the late afternoon … You're gonna do a check-in and this is gonna cause anybody who recompiles that day have to watch their computer for 45 minutes (…) and most of the time, you're gonna see this coming at 2 or 3 in the afternoon, you don't see folks (….) you don't see people doing* [file 1] *or* [file 2] *checking-in at 8 in the morning, because everybody all day is gonna sit and recompile."*

The transition from private work, then, is recognized as a point at which the work of a single developer can impact the work of others. Developers' orientation is not simply towards the artifacts but towards the work of the group. The subtlety with which the transition is managed reflects this consideration.

5.3 Public Work

The work of one developer becomes public when it is visible to all other co-workers. This happens in two different circumstances: when a developer changes the settings of his workspaces to grant others access to his code and after a developer commits his changes into the repository of the CM tool. These situations raise the question of how the MVP developers handle the new public work (changes)?

In the former case, the work is public but not shared, which means that it is not going to affect other developers' work. Therefore, MVP developers do not need to take any step in order to handle the public work, because it will not affect them. However, in the second case, MVP developers might need to adapt their work based on these changes. Indeed, MVP developers might need to recompile their changes (work) in case they choose to incorporate the new public work or they might need to change the run-time parameters of a process that was altered by the changes. Based on our data, we found out that the configuration management tool provides *some* help to MVP developers handle this situation. As mentioned before, these tools have building mechanisms that help MVP developers, upon request, to incorporate the new changes and identify syntactic conflicts between the developer's 'work-in-progress' and the new changes. However, these tools are not able to detect semantic conflicts since they are purposely created to be independent of programming languages [11].

The bug tracking tool, on the other hand, provides support for public work because all the operations performed in the problem reports are automatically visible to all MVP developers. In addition, this tool implements some accounting features that record the history of a PR including all operations performed on each one of them.

5.4 Moving from Public to Private Work, or "Breaking the code"

According to Walrad and Strom [33], the branch-by-purpose strategy adopted by the MVP team (see section 3.3) assures continual integration of the code, therefore minimizing problems. However, this strategy needs to be complemented by some form of notification that informs all developers that a check-in happened (and therefore that some integration took place). As mentioned before, this is achieved in the MVP team through the e-mail notification sent before the check-in's. Therefore, whenever a new change is introduced in the repository, all developers are notified about it. This affords an easy detection of problems caused by the introduced changes. In other words, if a change introduces a bug in the software, other developers might be able to detect it because: (i) they are aware that a change was introduced in the code by another developer; and (ii) they usually integrate the new introduced changes in their own work. If any abnormal behavior is identified in the software after a check-in, whoever identified that will contact the author of the check-in to verify if the problem is happening because of the check-in. If that is the case, the software is called "broken" and the code that was checked-in must be removed from the repository, corrected, and checked-in again later. In other words, the publicly available work needs to be made private again. The CM tool supports this transition because

it provides rollback facilities that allow one to remove committed changes from the repository.

6. DISCUSSION

The notions of private and public work and workspaces are well known ones in the design of collaborative systems. However, our empirical observations draw attention to the complex set of practices that surround the *transition* between public and private. Private information has public consequences, and vice versa.

The different formal and informal work practices arise in the MVP team, especially, because of the interdependencies among the different artifacts created during the software development process. Indeed, these interdependencies make the process of publicizing work so important. A developer can not simply carelessly publicize his work, because this will cause a large impact in other developers' work: some of them will need to go through their testing again, others will spend a lot of time recompiling their changes, others can need to change their own code in order to adapt the new checked-in code, and so on.

Since the MVP developers are aware of some of these interdependencies, they explicitly work to minimize problems that emerge in the relationship between their different working needs. Artifacts such as problem reports facilitate the management of interdependencies of developers from the different groups and with different roles. Problem reports are "boundary objects" in the sense of Star and Griesemer [29]; objects whose common identity is robust enough to support coordination, but whose internal structure, meaning, and consequences emerge from local negotiations between groups. Viewing PR's as boundary objects draws attention to their role in managing loosely-coupled coordination, and how each developer is able to interpret the information in the PR's that is useful to their current work. Critically, this is achieved without changing the identity of each PR along the whole software development process. Indeed, each PR keeps the same unique identifier.

Interestingly, these formal and informal work practices require that the author of the changes performs *most* of the additional work. However, this author will not get any benefit from that. Indeed, sending e-mail notifications, holding check-in's, and filling the appropriate PR's fields during the implementation are all operations performed by the author of the changes and none of them facilitate or improve his work. There is one developer performing the extra-work who does not gain any benefit of this extra work, and fifteen other developers who benefit from his work[3]. That is exactly one of the situations that lead groupware applications to fail [15]. In this particular software development team though, this does not happen. MVP developers are aware of the extra-work that they need to perform, but they are also aware that this same extra-work is going to be performed by the other developers when necessary, and this is going to help each and every one of them in performing their tasks.

On the other hand, MVP developers also adopt informal practices during their private work. The first one, called "partial check-in's", is especially important because it is used to handle files with a high degree of parallel development and changes in these files positively correlate with the number of defects [23]. "Partial check-in's" are variations of the formal software development process, which establishes that check-ins only will be performed when the entire work is done. They are necessary because of the software development tools adopted are unable to properly handle merging in these files. This is the same reason, according to Grinter [14], that led other team of software developers to either avoid parallel development or rush to finish their work. On the other hand, MVP developers rush because they do not want to repeat their testing when another developer checks-in some code into the repository. In both studies, developers describe their dilemma: they want to produce high-quality code, but they also want to finish fast their changes.

Holding onto check-in's is another informal approach adopted by the MVP developers during their private work. It is adopted because they are aware of some of the existing interdependencies in the software and they want to minimize the impact that their changes will cause on others' work. To be more specific, they understand that some changes cause a lot of recompilation, which might lead other developers to spend time "watching" the recompilation.

All this extra-work performed by the different members of the MVP team is another form of articulation work [27] that occurs in cooperative software development. It is different from the recomposition work [13], which is the coordination required to assemble software development artifacts from their parts. Recomposition work focuses on choosing the right components to create a software artifact due to source-code dependencies, while this extra work that we report focuses on the management of all interdependencies that exist in a software development effort.

After any code is checked-in into the CM repository, the other MVP developers are able to detect problems, or, detect if the MVP software is "broken". As noted in other settings such as ship bridges [19] or aircraft cockpits [20], this can be achieved because work artifacts and activities are visible to all. By creating a public space, the CM repository supports collective error detection and correction.

7. IMPLICATIONS FOR TOOLS

Software engineers have been developing tools to help co-workers in analyzing the impact of others' work in their own work. In this case, the support is provided to the developers after the transition from private to public work has been made. This approach, called change impact analysis [3], uses several techniques. One example is dependency graph approaches, which focus on determining the impact of the changed code (product) in other's part of the source code. These approaches are usually based on program dependences, which are syntactic relationships between the statements of a program representing aspects of the program's control flow and data flow [24]. In other words, they focus only in determining the impact of the changes in the product in the rest of the cooperative effort. Although powerful, these techniques are also computationally expensive and very time-consuming to be used by developers in their daily work. Consequently, they do not completely support the transition from private to public work, and as we've seen, this is a very subtle step in cooperative software

[3] The MVP group is composed of 16 developers. One of them is performing the check-in; therefore 15 others are being helped by the extra-work.

development. Although these techniques have their limitations, they are evidence that the dependencies between developers' working activities are a cause for concern and attention. We argue that other cooperative efforts, especially those with several interdependencies, could greatly benefit from such approaches, if they were arranged to support the emergence of public information.

Recent approaches in software engineering attempt to provide useful information to developers so that they can better coordinate. In other words, these approaches try to increase the awareness [7] of software engineers about the work of their colleagues. They differ, however, on the type of information that is provided. A first approach is based on the idea of facilitating the dissemination of public information by collocating software developers in warrooms [32]. In this case, companies expect to achieve the same advantages that the public availability of others' actions has brought to other settings such as ship bridges [19], aircraft cockpits [20], transportation control rooms [17] and city dealing rooms [16]. Indeed, early results of this approach have been encouraging [32]. However, there are practical limitations in the size of the teams that can be collocated, which suggests that tool support is still necessary. Indeed, new tools like Palantir [25] and Night Watch [22] adopt a different approach that focuses on constantly publicizing information(like CM commands) collected from a CM workspace to other workspaces that are accessing the same files. In this case, instead of focusing in the transition between private and public aspects of work, these tools basically eliminate the private work by making all aspects of the work publicly available to others. However, as discussed in section 2, the need for privacy and for controlling the release of private information is an important aspect in any social setting; which therefore needs to be addressed in the design of cooperative tools.

Finally, our data suggests that a software developer might use different sources of information at different times in order to assess the current status of the work. As mentioned before, the MVP team uses information from e-mail messages, the configuration management tool and the bug tracking system. By reading e-mail, MVP developers are aware of future changes in the CM tool because somebody else is going to check-in something. By inspecting only the CM tool, a developer can be aware of partial check-ins in the repository that are not reported by e-mail. And finally, the bug-tracking tool, through its PR's, provides information about how a developer's work is going to be impacted by the problem report associated with the check-in. These are three different tools that a MVP developer has to use. We believe that a possible improvement is to use event mechanisms (such as event-notification servers) to integrate these different sources of information, and then provide a unique interface and tool to assess the relevant information. Furthermore, abstraction techniques [18] could be employed to generate high-level information (e.g., status of the work) from low-level information like recent check-ins and check-outs, e-mails exchanged among software developers, information added to the bug-tracking tool, etc. This is an interesting research area that we plan to explore.

8. CONCLUSIONS AND FUTURE WORK

This paper examined the transitions between private and public work based on empirical material collected from a large-scale software development effort. The team studied, called MVP, uses mostly three tools to coordinate their work: a configuration management (CM) tool, a bug-tracking system, and e-mail. These tools provide support for private and public work, as well as some technical support that facilitates the transition from the former aspect to the latter. However, MVP developers also adopted a set of formal and informal work practices to manage this transition. These transitions are necessary to facilitate the management of the interdependencies among the different software artifacts. The following practices were identified and described in the paper: partial check-in's, holding onto check-in's, problems reports crossing team boundaries, code and design reviews, "speeding-up" the process, and finally, the convention of adding the description of the impact of the changes in the e-mail sent to the group. These practices suggest that analytical attention needs to be given to these transitions in order to enhance our understanding of cooperative work. Furthermore, computational support also needs to be provided so that this task can occur properly.

We plan to study other software development teams in order to understand how they deal with the aforementioned transition and their work practices to perform that. By doing that, we expect to learn important characteristics that can help us in understand other cooperative efforts.

9. ACKNOWLEDGMENTS

The authors thank CAPES (grant BEX 1312/99-5) and NASA/Ames for the financial support. Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Funding also provided by the National Science Foundation under grant numbers CCR-0205724, 9624846, IIS-0133749 and IIS-0205724. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

10. REFERENCES

[1] Ackerman, M. S., "The Intellectual Challenge of CSCW: The Gap Between Social Requirements and Technical Feasibility," *Human-Computer Interaction*, vol. 15, pp. 179-204, 2000.

[2] Appleton, B., Berczuk, S., et al., "Streamed Lines: Branching Patterns for Parallel Software Development," vol. 2002, 1998.

[3] Arnold, R. S. and Bohner, S. A., "Impact Analysis - Towards a Framework for Comparison," International Conference on Software Maintenance, pp. 292-301, Montréal, Quebec, CA, 1993.

[4] Bowers, J., "The Work to Make the Network Work: Studying CSCW in Action," Conference on Computer-Supported Cooperative Work, pp. 287-298, Chapel Hill, NC, USA, 1994.
[5] Conradi, R. and Westfechtel, B., "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, pp. 232-282, 1998.
[6] Curtis, B., Krasner, H., et al., "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, pp. 1268-1287, 1988.
[7] Dourish, P. and Bellotti, V., "Awareness and Coordination in Shared Workspaces," Conference on Computer-Supported Cooperative Work (CSCW '92), pp. 107-14, Toronto, Ontario, Canada, 1992.
[8] Dourish, P. and Bly, S., "Portholes: Supporting Distributed Awareness in a Collaborative Work Group," ACM Conference on Human Factors in Computing Systems (CHI '92), Monterey, CA, 1992.
[9] Ellis, C. A., Gibbs, S. J., et al., "Groupware: Some issues and experiences," *Communications of the ACM*, vol. 34, pp. 38-58, 1991.
[10] Erickson, T. and Kellogg, W. A., "Social Translucence: An Approach to Designing Systems that Support Social Processes," *Transactions on HCI*, vol. 7, pp. 59-83, 2000.
[11] Estublier, J., "Software Configuration Management: A Roadmap," Future of Software Engineering, pp. 279-289, Limerick, Ireland, 2001.
[12] Grinter, R., "Supporting Articulation Work Using Configuration Management Systems," *Computer Supported Cooperative Work*, vol. 5, pp. 447-465, 1996.
[13] Grinter, R. E., "Recomposition: Putting It All Back Together Again," Conference on Computer Supported Cooperative Work (CSCW'98), pp. 393-402, Seattle, WA, USA, 1998.
[14] Grinter, R. E., "Using a Configuration Management Tool to Coordinate Software Development," Conference on Organizational Computing Systems, pp. 168-177, Milpitas, CA, 1995.
[15] Grudin, J., "Why CSCW applications fail: Problems in the design and evaluation of organizational interfaces," ACM Conference on Computer-Supported Cooperative Work, pp. 85-93, Portland, Oregon, United States, 1988.
[16] Heath, C., Jirotka, M., et al., "Unpacking Collaboration: the Interactional Organisation of Trading in a City Dealing Room," Third European Conference on Computer-Supported Cooperative Work, pp. 155-170, Milan, Italy, 1993.
[17] Heath, C. and Luff, P., "Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Control Rooms," *Computer Supported Cooperative Work*, vol. 1, pp. 69-94, 1992.
[18] Hilbert, D. and Redmiles, D., "An Approach to Large-scale Collection of Application Usage Data over the Internet," 20th International Conference on Software Engineering (ICSE '98), pp. 136-45, Kyoto, Japan, 1998.
[19] Hutchins, E., *Cognition in the Wild*. Cambridge, MA: The MIT Press, 1995.
[20] Hutchins, E., "How a Cockpit Remembers its Speeds," *Cognitive Science*, vol. 19, pp. 265-288, 1995.
[21] Mark, G., Fuchs, L., et al., "Supporting Groupware Conventions through Contextual Awareness," European Conference on Computer-Supported Cooperative Work (ECSCW '97), pp. 253-268, Lancaster, England, 1997.
[22] O'Reilly, C., Morrow, P., et al., "Improving Conflict Detection in Optimistic Concurrency Control Models," 11th International Workshop on Software Configuration Management (SCM-11), Portland, Oregon, 2003 (to appear).
[23] Perry, D. E., and, H. P. S., et al., "Parallel Changes in Large-Scale Software Development: An Observational Case Study," *ACM Transactions on Software Engineering and Methodology*, vol. 10, pp. 308-337, 2001.
[24] Podgurski, A. and Clarke, L. A., "The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance," Symposium on Software Testing, Analysis, and Verification, pp. 168-178, 1989.
[25] Sarma, A., Noroozi, Z., et al., "Palantír: Raising Awareness among Configuration Management Workspaces," Twenty-fifth International Conference on Software Engineering, pp. 444-453, Portland, Oregon, 2003.
[26] Schmidt, K., "The critical role of workplace studies in CSCW," in *Workplace Studies : Recovering Work Practice and Informing System Design*, P. Luff, J. Hindmarsh, and C. Heath, Eds.: Cambridge University Press, 2000, pp. 141-149.
[27] Schmidt, K. and Bannon, L., "Taking CSCW Seriously: Supporting Articulation Work," *Journal of Computer Supported Cooperative Work*, vol. 1, pp. 7-40, 1992.
[28] Sellen, A. J. and Harper, R. H. R., *The Myth of the Paperless Office*. Cambridge, Massachusetts: The Mit Press, 2002.
[29] Star, S. L. and Griesemer, J. R., "Institutional Ecology, Translations and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology.," *Social Studies of Science*, vol. 19, pp. 387-420, 1989.
[30] Stefik, M., Foster, G., et al., "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings," *Communications of the ACM*, vol. 30, pp. 32-47, 1987.
[31] Strauss, A. and Corbin, J., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, Second. ed. Thousand Oaks: SAGE publications, 1998.
[32] Teasley, S., Covi, L., et al., "How Does Radical Collocation Help a Team Succeed?," Conference on Computer Supported Cooperative Work, pp. 339-346, Philadelphia, PA, USA, 2000.
[33] Walrad, C. and Strom, D., "The Importance of Branching Models in SCM," *IEEE Computer*, vol. 35, pp. 31-38, 2002.
[34] Whittaker, S. and Schwarz, H., "Meetings of the Board: The Impact of Scheduling Medium on Long Term Group Coordination in Software Development," *Computer Supported Cooperative Work*, vol. 8, pp. 175-205, 1999.